Introduction

A computer system is comprised of a collection of:

Processing elements - perform transformations on data items

CPU (*central processing unit)*

- performs arithmetic operations on information stored in memory
- controls movement of information within main memory

GPU (*graphics card)*

- converts graphics commands to pixel data
- modern GPUs also work like parallel CPUs

Disk controller

- controls the transfer of data between the PCI bus and disk
- computes error correction code (ECC) values

Storage elements – store data items for later use

Cache memory

- very fast "scratchpad" memory used for short term (~ 1 second) storage
- volatile (contents lost at power off)

Main memory

- very large but less fast than cache
- also volatile

Disk

- very very large but *WAY SLOWER* than main memory
- non-volatile (unless you drop the computer on the floor!)

Communication elements – transfer data items between processing and storage elements

Internal Processor bus

- CPU to cache

PCI Bus

- CPU to memory and controllers (graphics, disk, etc)

SCSI Bus

- Controller to devices disk, tape, etc.

Ethernet link

- Computer to another computer

802.11 link

- Wireless connection to another computer

Programming a computer

We will consider a simple hypothetical “human powered” computer with a CPU (the human) and a main memory consisting of a wall of “post office” type boxes. The human computer is limited his capacity to understand instructions but performs the instructions he can understand with 100% reliability.

The *memory* is conceptually modeled as a "post office box" system. Each box has a unique address or “box number”. Addresses range from 0, 1, 2, ... , n-1 where n is the number of different boxes.

Each box can hold exactly one slip of paper which contains either:

- An instruction to the human or
- An integer number

Fundamental memory behavior

Memory in both real and hypothetical computers adheres to two fundamental principles.

- The content of a memory "box" may be unknown to you but is *NEVER UNDEFINED.*
- When a memory "box" has been written into, it will maintain the value written until re-written.

Therefore, a memory "box" always contains the *last value* written to it.

The human computer instruction set

store value,box#

store 130,101 *! Stores the value 130 in box # 101*
store 10,102 *! Put a 10 in box #102*

add box#,box#,box#

add 101,102,110 *!Adds the values stored in box 101 and 102 and puts the answer in box 110*

This does NOT mean that the value 101+102 = 203 goes into box #110! We can't know what will go into box #110 unless we know the value presently stored in box #101 and box #102. Since we already stored 130 in box #101 and 10 in box #102, we may infer that the value 140 will be stored in box 110.

sub box#,box#,box#

sub 100,200,300 *!Subtracts the value stored in box 200 from the value stored in box 100 and puts the answer in box 300*

halt *!Program is complete, stop*

When the human computer is told to *start* he fetches his first instruction from box 0 and then proceeds sequentially through box 1, box 2, ... until he encounters a *halt* instruction

The missing ingredients

What we have so far is like early, crude spreadsheet macro languages... somewhat useful but missing two key ingredients (both of which are present in modern spreadsheet macro languages).

These ingredients are:

- conditional execution of a group of instructions
- repeated execution of a group of instructions

Adding the missing ingredient

When the human computer is told to start, he fetches the first instruction from box 0 and then proceeds sequentially through box 1, box 2, ... until he encounters a halt instruction.

The missing ingredient is the *jumpc* (jump conditional) instruction which tells the human that

- if a specified condtion is true, then the normal sequential execution should be altered and the next instruction fetched from a specified box.

- if the condition is false, sequential execution should proceed as normal

jumpc box#,op,box#,box#

jumpc 100,lt,101,10 !if the value stored in box 100 is less than the value stored in box 101 then fetch your next instruction from box 10. If the instruction in box 10 is neither halt nor jumpc the instruction fetched after box 10 will be in box 11.

Other conditions include

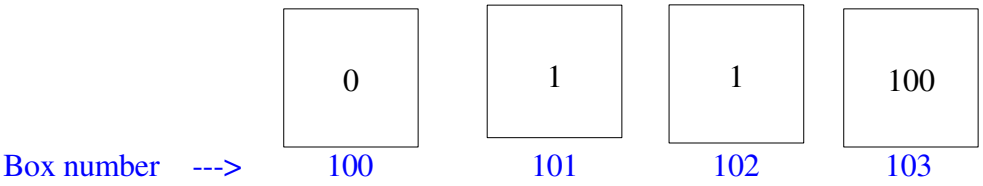
ne – not equal
eq – equal
le – less than or equal to
gt – greater than
ge – greater than or equal to

With the addition of *jumpc* any result that can be computed by any computer can be obtained (albeit slowly) by the human computer.

Adding the 1st 100 integers

Problem: Add 1+2+3+....+100 and leave the sum in box 100.

```
Box
 0      store 100,103          !Store the upper limit of 100 in box 103
1       store 0,100            !Initialize the sum of all values added so far to  0
2       store 1,101            !Initialize the value to be added to be 1 initially
3       store 1,102            !We also need a value to increment the value to be added
4       add 101,100,100        !add contents of box 101 to box 100 leaving sum in 100
5       add 102,101,101        !increment the value to be added to box 100 by 1
6       jumpc 101,le,103,4      !as long as the value in box 101 is less than or equal to the 100 in box
                               !103 continue to add
7       halt
```



Exercises: Which boxes hold values that never change?
What values will be in box 100 and 101 just before the third time the instruction in box 4 is executed?
The human computer doesn't have a multiply instruction. Write a program to multiply the value in box 100 by the value in box 101 leaving the result in box 102.

The Algorithm

This hypothetical program is written in a symbolic "machine language" sometimes called Assembly Language. As stated previously, it is the case that any result that may be computed on any computer may be computed (though much more slowly!) by the human computer. Thus, writing other sample programs for the human computer is a useful exercise as it provides useful practice in the art of specifying an *algorithm*[1] precisely and correctly.

The human computer also illustrates an extremely important concept: the distinction between the address of a memory cell (100) and the value it contains (1 + 2 + ... + 100)

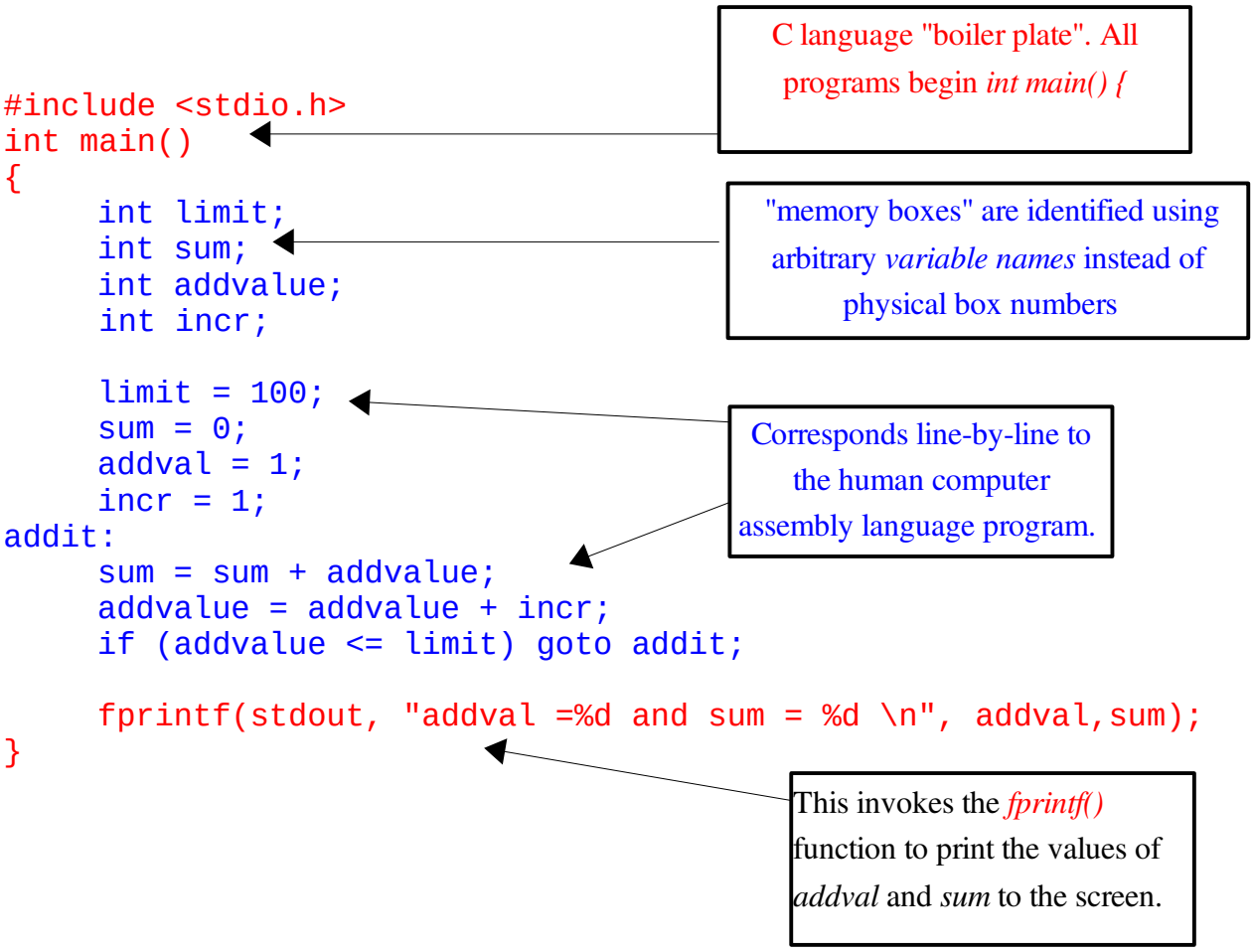
In summary

Computer hardware can perform simple operations (add, multiply, subtract, etc.) and perform comparison tests very fast and very reliably. Nevertheless, it doesn't possess even rudimentary problem solving skills. It doesn't do algbra or calculus. It can only execute a prescribed set of simple operations (a program!) with great speed and accuracy. Thus, the true power of the computer is supplied by its human programmer!!

1 Although he is reputed to have invented both the Intenet and Global Warming, rumors that Al Gore also invented the Algorithm are categorically *false.*

The human computer language and the C programming language:

C is sometimes referred to has a “high level” assembly language because, of all modern programming languages, C programs map most directly and naturally to the machine level. We can write the program that sums the first 100 integers in C as follows:




```
#include <stdio.h>
int main()
{
     int limit;
     int sum;
     int addvalue;
     int incr;

     limit = 100;
     sum = 0;
     addval = 1;
     incr = 1;
addit:
     sum = sum + addvalue;
     addvalue = addvalue + incr;
     if (addvalue <= limit) goto addit;

     fprintf(stdout, "addval =%d and sum = %d \n", addval,sum);
}
```

Note that in C, a variable name instead of an actual address is used to identify the box in which the variable resides, but the program itself is line by line equivalent.

A computer program that was capable of translating the C code above to Assembly Language is called a compiler.

Representing Information within a Computer System

We will be concerned with representing three basic types of information in computer system

- Integer (whole) numbers
    - (signed (positive and negative: 14, -42) and
    - unsigned(non-negative only: 0, 13))

- Floating point numbers (contain a fractional part optionally written in in scientific notation e.g., 1.3 or 13e-1)

- The “Latin” character set in which English is written

Before turning to the details of the representation, we will consider the organization of the computer memory system in which the information is stored.

Computer Memory

Computer memory is a comprised of a large collection of two-state (off/on) electrical devices called bits (binary digits). A single electronic bit can assume two distinct physical states which are commonly called {0 , 1}.

Because a single bit encodes so little information, to solve useful problems it is necessary to aggregate bits into larger elements.

With two bits we can encode 4 distinct values, {00, 01, 10, 11}. This might lead one to think that the number of distinct values that may be encoded is 2 x the number of bits, *but that would be incorrect.*

With three bits we can encode not 6 but 8 distinct values {000, 001, 010, 011, 100, 101, 110, 111}. In general, with n bits we may repesent 2^n distinct values or "elements of information".

Encoding schemes used in computer programs often use *completely arbitrary encodings* to represent information in different domains.

For example, common house pets might be encoded:

00 – dog
01 – cat
10 – Vietnamese pot bellied pig
11 – bird

Modern computer memory organization

A computer memory is one dimensional array of individually addressable storage elements (analogous to the post office boxes of the human computer).

For reasons discussed on the previous page, it was decided in the design of the very earliest computers that each "addressable box" must contain *more than* 1 bit of information.

The basic addressable unit of memory

The "optimal" number of bits in a "box" is arbitrary and various values have been tried throughout computer history. Using the term byte to mean 'the basic addressable unit of memory', 5, 6, 7, 8, and 9-bit bytes have all been used. Other early systems designed for scientific computation eschewed the byte altogether and organized their memories using words containing up to 60+ bits.

Based upon

- the success of the IBM System 360 (1960's) computer system,
- and the recognition that life was good when the number of bits in byte is a power of 2

in virtually all modern computer systems a byte is comprised of 8 bits. Half of a byte (4 bits) is sometimes called a nibble.

Possible values of a byte of memory

A single 8-bit byte can encode = 2^8 = 256 distinct values or elements of information:

00000000 - 0
00000001 - 1
00000010 - 2
00000011 - 3
:
:
11111110 - 255
11111111 - 256

Addresses contrasted to contents

It is extremely important to understand and distinguish

the address of a memory element (the "box number")
the contents of a memory element (the contents of the box)

Addresses -

begin at 0 and increase in unit steps to N-1 where N is the total number of bytes in the memory space.

Contents -

Since each basic storage element consists of only 8 bits, there are only 2^8 = 256 different values that can be contained in a single byte storage element.

Positional number systems

We have seen that numbers are stored in a computer memory using a fixed number of binary digits or bits to encode each value.

Accordingly, computers *perform integer arithmetic in base 2.*

Numbering systems are called positional when the *position* of a particular "digit" within a number determines the digit's contribution to the final value of the number. In the number 666 the first 6 represents 600, the second 6 represents 60, and the final one represents 6.

*Roman numeral*s provide an example of a *non-positional* system. The "digit" V means 5 regardless of where it appears in the number. Doing arithmetic is non-positional systems is very challenging.

Experiments: Convert the Roman number XLVII to decimal. Multiply XLV by CIV.

Binary (base 2) and decimal (base 10) arithmetic

Binary or base 2 arithmetic is a positional system that works analogously to base 10 but uses 2 rather than 10 distinct "digits". (The choice of base 10 in "human" arithmetic was arbitrary and based upon the number of fingers (digits) possessed by the average human.)

In base 10 the number 1234 means 1 x 10^3 + 2 x 10^2 + 3 x 10 ^ 1 + 4 x 10 ^0 or

1000
200
30
+ 4
1234

Thus a digit at position n (n = 0, 1, 2, 3, ...) is implicitly understood to be multiplied by $10 \wedge n$.

Binary representation

In the binary system, the bit at position *n* is implicitly understood to be multiplied by *2 ^ n* (instead of 10 ^ n), and there are only two digits (bigits?).

Analogous to base 10, each power of two value (*2 ^ n*) starts with a 1 bit and is followed by *n* 0's. Powers of two (written in decimal) are: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536. Written in binary these values are 1, 10, 100, 1000, 10000, 100000, 1000000, etc..

Thus, the binary number 10110 means 1 * 2 ^ 4 + 0 * 2 ^ 3 + 1 * 2 ^ 2 + 1 * 2 + 0

16 = 1 * 2 ^ 4
0 = 0 * 2 ^ 3
4 = 1 * 2 ^ 2
2 = 1 * 2 ^ 1
+0 = 0 * 2 ^ 0
22 base 10 = 10110 base 2

and we have just devised an *algorithm* for converting from base 2 to base 10!

Exercise: Convert 1101 and 11011101 from base 2 to base 10.

Counting in base 2 and in base 10

In decimal counting each time the low order digit exceeds 9 it is reset to 0 and a 1 is "carried" into the next column (possibly causing another reset to zero and carry.)

Binary counting work analogously, but the reset and carry occurs each time a digit exceeds 1!

Binary	*Decimal*
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15
10000	16
10001	17

Note that as we count in binary the low order digit assumes values 0,1,0,1,0,1.

The second digits assume values	0,0,1,1,0,0,1,1
The third digits assume values	0,0,0,0,1,1,1,1
The fourth digits assume values	0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1

Exercises: Write the decimal value 13 in binary. Write the binary value 1011 in decimal.

Converting from base 10 to base 2.

One can convert from base 10 to base to by a sequence of divisions by 2.

- In each successive division the remainder is a bit in the base 2 representation.
- The quotient becomes the dividend in the next stage of the operation.
- The bits are produced from least significant (low order) to most significant.
- The procedure ends when the quotient becomes 0.

Example: Convert 67 base 10 to base 2.

67 / 2 = 33 r 1 <-- least significant (2 ^ 0) bit
33 / 2 = 16 r 1
16 / 2 = 8 r 0
8 / 2 = 4 r 0
4 / 2 = 2 r 0
2 / 2 = 1 r 0
1 / 2 =- 0 r 1 <-- most significant (2 ^ 6 = 64) bit

Thus 67 base 10 = 1 0 0 0 0 1 1 base 2

One can verify the accuracy of the computation by converting the result back to base 10.

Here: 64 + 2 + 1 = 67

Exercise: Convert 93 base 10 to base 2.

Bases other than 2 and 10

Any positive integer can be used as a base. For example, in base 3

- The "digits" (thrigits?) are {0, 1, 2}
- In a number, such as 12012, each digit is implicitly multiplied by a positional power of 3: 1, 3, 9, 27, 81, 243, ...

2 x 3 ^ 0 = 2
1 x 3 ^ 1 = 3
0 x 3 ^ 2 = 0
2 x 3 ^ 3 = 54
+1 x 3 ^ 4 = 81
140 base 10

Conversion from base 10 to base 3 is accomplished by a sequence of divisions by 3

49 / 3 = 16 r 1
16 / 3 = 5 r 1
5 / 3 = 1 r 2
1 / 3 = 0 r 1

Answer: 1211 = 1 * 3^3 + 2 * 3^2 + 1 * 3 + 1 = 27 + 18 + 3 + 1 = 49

Exercises: Convert 134 base 5 to base 10. Convert 79 base 10 to base 4.

Bases that are powers of 2

Binary numbers are very difficult for humans to write and remember:
Consider the 32 bit number:

10110101101110111101010110111110

Bases that are powers of 2 are useful in simplifying notation.

A single digit of a base 2^n system represents exactly n bits of a base 2 system.

Base	# of bits	
4	2	*tetral?*
8	3	*octal*
16	4	*hexadecimal*

Converting from base 2 to base 4: (valid digits (0, 1, 2, 3))

Base 2: 10 11 10 01
Base 4: 2 3 2 1

Converting from base 2 to base 8 (valid digits (0, 1, 2, 3, 4, 5, 6, 7))

Base 2: 101 110 001 010
Base 8: 5 6 1 2

Base 16 – *Hexadecimal*

Since 16 is greater than 10, there are not enough digits in the base 10 system to encode all of the single digit base16 numbers. Thus we augment the digits 0-9 with the first 6 letters of the alphabet.

Each base base 16 value corresponds to a 4 bit binary number as shown here:

Base 16 digits: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

binary	*hex*	*binary*	*hex*
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

A single 8 bit byte can always be encoded in exactly 2 hexadecimal digits:

Base 2	Base 16
1001 1011	9B

The 9 is called the high order nibble aand the B is the low order nibble.

An 32 bit integer can be converted by grouping the 32 bits into eight 4 bit nibbles:

1011	0101	1011	1011	1101	0101	1011	1110
B	5	B	B	D	5	B	E

Exercise: Convert 010101110111110 to hexadecimal. Convert A12BC to binary.

Representing numeric values

More than 8 bits are needed for useful application to numerical problems. Thus it is common to group adjacent bytes into units commonly called words.

- Multiple word lengths are common, and common word lengths include 2 bytes, 4 bytes, and 8 bytes.
- In some architectures (Sun Sparc) it is required that the address of each word be a multiple of word length. That is, the only valid addresses of 4-byte words are 0, 4, 8, 12, 16, ...)
- In other architechtures, words can reside on any byte boundary, but the use of unaligned words often causes a performance problem.
- Common word lengths now include 16, 32, 64 and 128 bits which correspond to 2, 4, 8, and 16 bytes per word.
- Word lengths may also differ between integer and floating point data types.
- Even when bytes are aggregated into words, addresses remain byte oriented in modern computer systems.
- The addresses of 4 byte words are thus 0, 4, 8, 12, 16, ...

Nevertheless, even with these aggregation strategies computer arithmetic is not the same as true mathematical arithmetic.

Mathematical integer arithmetic

Mathematically, the integers consist of a countably infinite set of values:

$$-\infty, \ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots, \infty$$

The integers comprise what is called a ring.

- The sum of two integers is an integer
- The difference of two integers is an integer
- The product of two integers is an integer
- The division of one integer by another is not defined.

Computer integer arithmetic

Computer arithmetic is an approximation of mathematical arithmetic

- The number of distinct integers is 2^{word_length} where word length = 8, 16, 32, 64, bits
- Unlike true integer arithmetic on the computer the number of distinct values is finite.
- If word length is 8 bits using signed integer arithmetic you can represent only the values -128, ... -1, 0, 1, ... 127
- If you compute 100 + 100 you get the wrong answer because the maximum positive number that you can represent with 8 bits of information is 127
- A computer instruction set supports integer division.
- If you compute 100 / 9 you get an approximate answer. Although computer systems do support integer division, they do so by discarding any remainder
- So 100 / 9 = 11 (instead of 11.111111.....), and 2 / 5 = 0 (instead of 0.4).

Computer integer arithmetic

The principles of arithmetic work the same in any positional system. Consider the decimal addition:

```
  13
+  9
  22
```

We proceed from right to left and if a sum exceeds 9 then it is forced to "wrap around" and a carry into the next column is generated.

In binary 0 + 0 = 0; 0 + 1 = 1 + 0 = 1; and 1 + 1 = 0 with a carry out.

We will use 8 bit arithmetic as an example, but 16 and 32 bit arithmetic operates in essentially the same way.

```
  1000  0111    129
+ 0010  0101     33
  1010  1100    162
```

Finite value size and overflow

A mathematical integer consists of an unbounded number of bits ==> overflow can't happen.

A computer integer consists of a finite number of bits ==> overflow can happen.

8 bit *unsigned* arithmetic:

In unsigned arithmetic all values *are considered to by non-negative*.
Possible values of an 8 bit byte in base 10 are 0, 1, 2, ... 255 (0, 1, 10, ..., 111111111 base 2)

Consider what happens when we try to compute 240 + 32

```
  1111 0000   240
  0010 0000    32
  ---------------
  0001 0000    16
```

The 1 bit that carries out of the left end of the operation will be discarded and the answer we compute will be 16 which is (240 + 32) modulo 256 in mathematical terms. Computer systems typically provide low level mechanisms for detecting when these situations occur, but these mechanisms typically *are not available in high level* programming languages!!

Exercise: Add 01010110 + 11001000 in binary. Is the answer you get mathematically correct?

Signed 8 bit arithmetic -

For many useful applications its necessary be able to represent *negative numbers.*
Most computer systems now use a 2's complement representation for negative numbers.

- The 1's complement of a value is obtained by inverting all the bits.
- The 2's complement is then obtained by adding 1 to the 1's complement

0010 1101 -- base integer
1101 0010 -- 1's complement
1101 0011 -- 2's complement

- Since the negative of any number is its two's complement, the sum of a number and its two's complement is always 0.
- The difference a – b is computed as a + (-b) or a + twos_complement(b)
- Unlike “sign bit” systems, the twos complement system has only a single 0

Twos complement representation

Dec	Binary	Hexadecimal
-128	10000000	80
-127	10000001	81
-126	10000010	82
:		
-2	11111110	FE
-1	11111111	FF
0	00000000	00
1	00000001	01
2	00000010	02
:		
126	01111110	7E
127	01111111	7F

Overflow remains a problem

```
 96    01100000
+68    01001000
       10101000  --- which is a negative number!
```

Exercise: What is the twos complement of 10110000

Exercise: Write -17 as a twos complement signed integer

Encoding the alphabet

Its useful to encode *text* in computer memory and files:

- Names of account holders in financial records
- Text in word processors
- Part names in inventory systems, etc.

Although the encoding is arbitrary, there are some useful characteristics a code should possess:

- The letters are encoded sequentially: 'A' + 1 is 'B'
- Inverting a single bit can convert between upper and lower case

The code that we will be using is called ASCII (American Standard Code for the Interchange of Information). Printable characters start with the value hex 20 = 32 which is the code for blank space. Following space are many of the special characters that you find on the keyboard.

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2a	*
43	2b	+
44	2c	,
45	2d	-
46	2e	.
47	2f	/

The encoding of the character representation of the digits is in the range hex 30 to hex 39

Dec	Hex	Char
48	30	0
49	31	1
:		
56	38	8
57	39	9

More special characters follow in the range hex 3A to 40

Dec	Hex	Char
58	3a	:
59	3b	;
60	3c	<
61	3d	=
62	3e	>
63	3f	?
64	40	@

The upper case letters are next

Dec	Hex	Char
65	41	A
66	42	B
67	43	C
:		
88	58	X
89	59	Y
90	5a	Z

The ASCII encoding of the letter A is the byte having the value

$$0100\ 0001 = 2^{6} + 2^{0}$$

This value is written in decimal as 65 and in hexadecimal as 41

Another block of special characters occupy hex 5B to 60

Dec	Hex	Char
91	5b	[
92	5c	\
93	5d	]
94	5e	^
95	5f	_
96	60	`

The lower case characters occupy hex 61 to 7A.

Upper case A 0100 0001
Lower case a 0110 0001

Case conversion may be accomplished by inverting the bit in the red position

```
Dec  Hex Char
 97 61 a
 98 62 b
 99 63 c
   :

120 78 x
121 79 y
122 7a z
```

There are also a few special characters that follow *z*.

Exercise: What letter is represented in hex as 63? What is the binary encoding of the UPPER case version of the same letter.

Control characters:

The ASCII encodings between decimal 0 and 31 are used for to encode what are commonly called *control characters.* Control characters having decimal values in the range 1 through 26 can be entered from the keyboard by holding down the ctrl key while typing a letter in the set a through z.

Some control characters have "escaped code" respresentations in C, but all may be written in octal.

Dec	Keystroke	Name	Escaped code
4	ctrl-D	end of file	'\004'
8	ctrl-H	backspace	'\b'
9	ctrl-I	tab	'\t'
10	ctrl-J	newline	'\n'
12	ctrl-L	page eject	'\f'
13	ctrl-M	carriage return	'\r'

The EOF character has no \letter representation.

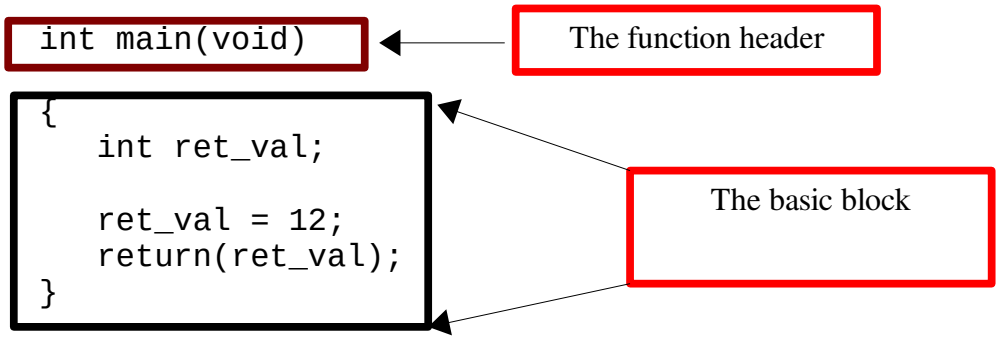
It may be expressed as '\004' or for that matter simply 4.

The structure of C programs

We will now present a *somewhat* formal view of the structure of a C program so that we don't have to learn *exclusively* by example. Learning from examples is very useful, but if we understand *the basic structure* of the language we can better understand *why* certain constructs are correct and others are not correct. This in turn allows us to be more efficient programmers.

- A C program consists of a collection of one or more *functions*.
- Each function must have a *unique name*.
- Exactly *one* of the function must have the name *main()*.
- When a program is run, execution starts at the beginning of the *main()* function.

Each function consists of a *header* followed by a *basic block*.



In the *function header*

- *int* defines the type of value returned by the function
- *main* is the name of the function
- the values enclosed in () identify any *parameters (variables)* that are being passed to the function. The reserved word *void* means there are *no parameters* will be used.

The *basic block* is delimited by { and } and consists of:

- *declaration of variables*
- *executable code*

Declaring variables of integer type in the C language

Before an integer variable can be used in a C program it must be declared via a statement of the following form:

optional_modifier type_name variable_name;

- The optional_modifier – The modifier is used to distiguish between signed and unsigned integers. The default modifier is signed . To create an unsigned integer the unsigned modifier must be explicitly provided.

- The type_name - The following type names create integer values.

`char -`	8 bit
`short-`	16 bit
`int -`	32 bit (sometimes 16)
`long-`	32 bit (sometimes 64)

Variable names:

- Comprised of upper and lower case letters, digits, and _
- Must start with a letter or _
- Must not be a reserved word (e.g. *int, while, char, unsigned* ....) See appendix A of the text
- Should not be the name of a standard function or variable.... but how do I know what those are?
- The name should be reflective of the variables' use in the program.

Any signed or unsigned integer variable can hold

- values used in arithmetic computation
- ASCII encoded representation of alphabetic and special characters

Neither the computer nor a human who examines the contents of memory can determine if a value <= hex 7F = 127 is being used as a number or as encoded character.

Example declarations:

```
unsigned char  red_pixel_value;
int            image_width;
unsigned short loop_counter;
long           image_size_in_bytes;
long           image_size_in_pixels;
char           first_letter;
```

Executable code

Executable code consists of a sequence of *statements.* Statements, themselves are constructed from *expressions*.

Expressions consist of (legal) combinations of:

- *operands* which may be further classified as
    - constants (integer, character, floating point)
    - variables
    - function calls

and

- *operators* which are discussed on the next page.

Operands -

Integer constants may be expressed as

- decimal numbers: 71 must not start with 0!!!
- octal (base 8) numbers: 0107 must always start with 0
- hexadecimal (base 16) numbers 0x47 must always start with 0x
- ASCII characters 'G' must be enclosed in single quotes

All of the above constants have the same value!

Different encodings may be freely intermixed in expressions

```
71 + 'G' + 0x47
```

(The value of the above expression is decimal 71 + 71 + 71 = 213)

Integer variables must be pre-declared as previously described *(unsigned) char, short, int, long*

```
unsigned char pixel;
```

Function calls consist of the name of the function being called along with a parenthesized collection of parameter values to be passed to the function:

```
how_many = fscanf(stdin, "%d", &input_value);
```

Operators **can be grouped into several classes.**

For now, we will be using only the ones shown in *red.*

Assignment: `=`

Assigns the value of the expression to the *right* to the *variable* on the *left.*

legal: `x = 4 + (3 + 2 * y)`

illegal: `x + (x - 3) = 17;`

Arithmetic: `+, -, *, /, %`

Add, subtract, multiply, divide

Comparative: `==, != , <, <=, >, >=`

equal, not equal, less than, less than or equal to, greater that, greater than or equal to

Logical: `!, &&, ||`

not, and, or

Bitwise: `~, &, |, ^`

not , and, or, exclusive or

Shift: `<<, >>`

Operators can also be characterized by the number of operands they support

All of the operators shown in *red* support two operands.. one on the left and one on the right.

```
x = 5          // assignment
y == 5         // comparision
z – 10         // subtraction
```

Some operators also support either one or two operands

```
-x             // negation
a - b          // subtraction
```

Others support only one operand

```
!x             // logical not
```

Expressions are built by combining operators and operands

Arbitrary sequences of operators and operands (e.g. x + - * y z / / =) are not meaningful and are thus not legal in the C language. In general we can say that

- *operands must be separated from each other by operators* and that
- *each operator must have the proper number (1 or 2) of operands*.

It may be difficult to determine the value of legal expressions such as:

```
v1 + 5 * v2 / 3 * v1
```

Suppose I tell you that the variable *v1* currently has value *4* and *v2* has value 3.
What is the value of the above expression?

The C compiler has a set of immutable rules for evaluating such expressions. The C compiler has a set of immutable rules for evaluating such expressions. The rules are called *precedence (e.g. mulitiply and divide are done before add and subtract) and associativity (e.g. a collection of multiply and divides is evaluated left to right so 42 / 6 * 7 = 49 and not 1),* but they are hard to remember.

We can ensure that the compiler does what we want by building our expressions from parenthesized sub-expressions. Such expressions are *always* evaluated *inside out*.

```
v1 + ((5 * v2) / (3 * v1))

v1 + (5 * (v2 / 3)) * v1

((v1 + 5) * v2) / (3 * v1)
```

An evaluation *tree*
Exercise: Create evaluation trees and find the value of each of the above three expressions.

Expressions that are *true* or *false*

In the C language

- an expression that *has the value 0* is *false*
- an expression that *does not have the value 0* is *true*

Statements

- An expression followed by a semi-colon is known as a *statement.*
- Here are some examples of statements:

The assignment statement

variable = expression;

computes the value of *expression* and stores the value in *variable*

x = y + z + 3;

The useless statement:

expression;

computes the value of *expression* and then *discards it.*

x + y – 2 / 3;

compares the value of two variables and then discards the answer

x == z;

Introduction to Input and Output

Useful computations often require a mechanism by which the user of a program may

- provide *input data* that is to be assigned to variables,

and another mechanism by which the program may

- produce *output data* that may be viewed by the user.

Initially, we will assume that the user of the program

- enters input data via the keyboard and
- views output data in a terminal window on the screen.

The C run time system automatically opens two files for you at the time your program start:

- *stdin* – Input from the keyboard
- *stdout* – Output to the terminal window in which the program was started

Eventually, we will see how to write and read files that reside on the computer's disk.

A simple C program

C programs consist of one or more *functions*, and every C program must have a main() function.
For now, it should be written as follows:

```
int main()
{
     return(0);
}
```

C programs often rely on functions that reside in the C run time libary. The functions perform input and output and other useful things such as taking the square root of a number.

When your function, uses functions in the run time libary, you must include proper header files. These header files:

- contain the declaration of variables you may need
- allow the compiler to determine if the values you pass to the function are correct in type and number.

If your program is to use the standard library functions for input and output, you must include the header file *stdio.h* as shown below.

```
#include <stdio.h>

int main()
{
     return(0);
}
```

Reading integer values from *stdin*

To read the value of a single integer into our program we expand it as follows:

```
#include <stdio.h>

int main()
{
    int howmany;    // how many integers were read
    int the_int;    // the integer value

    howmany = fscanf(stdin, "%d", &the_int);

    return(0);
}
```

Comments should be used to describe the use of variables

The *fscanf()* function returns the number of values it suceeded in reading. If *fscanf()* succeeds here, then howmany will be set to 1, because the statement is attempting to read only one value.

When a human enters numeric data using the keyboard, the values passed to a program are the ASCII encodings of the keystrokes that were made. For example, when I type:

```
261
```

3 bytes are produced by the keyboard. The hexadecimal encoding of these bytes is:

```
32 36 31   -  hex
 2  6  1   - ascii
```

To perform arithmetic using my number, *it must be converted to the internal binary integer representation.* Decimal 261 is binary 1 0000 0101 or 0x105 = 1 * 16 ^ 2 + 5.

Format conversion with *fscanf()*

```
howmany = fscanf(stdin, "%d", &the_int);
```

- The first value passed to the *fscanf ()* function is the identity of the file from which you wish to read. Note that stdin is not and must not be declared in your program. It is declared in <stdio.h>.

- The second parameter is the "format string". The "%d" format code tells the *fscanf()* function that your program is expecting an ASCII encoded integer number to be typed in, and that *fscanf()* function should convert the string of ASCII characters to internal 2's complement binary integer representation.

- The third value passed to *fscanf()* is the "box number" or address of the memory cell to which "the_int" was assigned by the compiler. The use of the & (address of) operator causes the compiler to pass the address of the box/memory cell rather than the value in the box/memory cell.

- The *fscanf()* function must be passed the address of *the_int* needs the address because fscanf must return the value to the memory space of the program. Passing in the value that is presently in the held int *the_int* would be useless.

- The value returned by *fscanf()* and assigned to the variable howmany is the number of values it successfully obtained. Here it should be 1. In general the number of format specifiers should always equal the number of pointers passed and that number will be returned at the completion of a successful scan.

Reading two ints in a single read

We can make a small modification to our program so that it now reads two distinct values from the standard input, *stdin.*

```
#include <stdio.h>

int main()
{
    int howmany;    // how many integers were read
    int another;      // a second input integer
    int the_int;    // the integer value

    howmany = fscanf(stdin,"%d %d", &the_int,
                          &another);

    return(0);
}
```

To do this *we must pass two format specifiers in the format string and pointers to the two variables* that we want to receive the two different values.

The number of values to be read must always match the format code. Here the value of *howmany* should be 2. If it is not 2, then a non-integer value or end of file was encountered in the input.

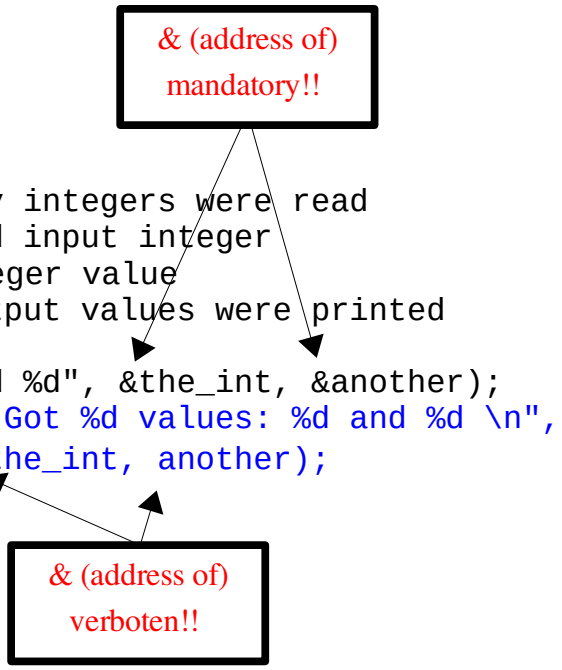
Formatted output with *fprintf()*

Since the program produces no output, its not possible to tell if it worked. We can obtain output with the *fprintf()* function.

```
#include <stdio.h>

int main()
{
     int howmany;    // how many integers were read
     int another;    // a second input integer
     int the_int;    // the integer value
     int printed;    // many output values were printed

     howmany = fscanf(stdin, "%d %d", &the_int, &another);
     printed = fprintf(stdout, "Got %d values: %d and %d \n",
                       howmany, the_int, another);
     return(0);
}
```



- The first parameter passed to fprintf() is the desired file. The file stdout, like stdin, is declared in `<stdio.h>`
- The second parameter is again the format string. Note that literal values that will appear in the output of the program may be included in the format string.
- The *%d* format code tells *fprintf()* that it is being passed the value of a binary integer.
- The fprintf() function will convert the binary integer to a string of ASCII characters and send the character string to the specified file.
- Note that the values of the variables are passed to *fprintf()*, but the addresses are passed to *fscanf()*! This is done because *fprintf()* doesn't need to store anything in the memory space of its caller.

Compiling and running the program:

```
/home/westall ==> gcc -g -Wall io.c
/home/westall ==> ./a.out
143 893
Got 2 values: 143 and 893
```

What happens if we provide defective input?

```
int /home/westall ==> ./a.out
12a 567
Got 1 values: 12 and 10034336
```

- *fscanf()* stops at the first bad character

- The value of *another was never set*, The value 10034336 is whatever was leftover in the "memory box" to which the variable *another* was assigned the last time it was used.

Processing multiple pairs of values

We now consider the problem of processing multiple pairs of numbers. Suppose we want to read a large collection of pairs of numbers and print each pair along with its sum.

As usual we start with a simpler prototype in which we try to process *a single pair.*

```
/* p2.c */

#include <stdio.h>

int main()
{
   int howmany;    // how many integers were read
   int num1;       // the first number of the pair
   int num2;       // the second number of the pair
   int sum;

   howmany = fscanf(stdin, "%d %d", &num1, &num2);
   sum = num1 + num2;
   fprintf(stdout, "%d  +  %d =  %d \n",
         num1, num2,  sum);
   return(0);
}
```

```
/home/westall/acad/cs101/notes06 ==> gcc -o p2 p2.c -g -Wall
/home/westall/acad/cs101/notes06 ==> p2

44 33
44  +  33 =  77
```

Extension to 4 pairs of numbers

We can easily extend our program to allow us to process 4 pairs

```
/* p3.c */
#include <stdio.h>
int main()
{
   int howmany;    // how many integers were read
   int num1;       // the first number of the pair
   int num2;       // the second number of the pair
   int sum;

   howmany = fscanf(stdin, "%d %d", &num1, &num2);
   sum = num1 + num2;
   fprintf(stdout, "%d  +  %d =  %d \n",
         num1, num2,  sum);

   howmany = fscanf(stdin, "%d %d", &num1, &num2);
   sum = num1 + num2;
   fprintf(stdout, "%d  +  %d =  %d \n",
         num1, num2,  sum);

   howmany = fscanf(stdin, "%d %d", &num1, &num2);
   sum = num1 + num2;
   fprintf(stdout, "%d  +  %d =  %d \n",
         num1, num2,  sum);

   howmany = fscanf(stdin, "%d %d", &num1, &num2);
   sum = num1 + num2;
   fprintf(stdout, "%d  +  %d =  %d \n",
         num1, num2,  sum);
   return(0);
}
```

```
/home/westall/acad/cs101/notes06 ==> gcc -o p3 p3.c -g -Wall
/home/westall/acad/cs101/notes06 ==> p3
1 3
1  +  3 =  4
3 4
3  +  4 =  7
5 6
5  +  6 =  11
7 8
7  +  8 =  15
```

Extension to arbitrary numbers of pairs to be added

- If we knew we had to process 1,000 pairs of numbers, we could duplicate the existing code 250 times!
- This approach has (hopefully obvious) disadvantages especially when extending to 10,000,000,000 pairs!

Controlling the flow of instruction execution

- The solution lies in mechanisms used to control the flow of instruction execution in a program.
- Such mechanisms permit us to instruct the computer to perform a task repetitively.

Recall the *human computer:*

- When an instruction had been executed,
- The human fetched his next instruction from the next (numerically) box.
- Non-human computers and high level languages work this way too!
- Both human and non-human computers provide a jumpc instruction
- If the condition tested is true then
    - the next instruction is fetched from a new specified location and
    - sequential execution resumes at the new specified location.

The *jumpc* instruction provides *all* the firepower that is needed with respect to control flow management to compute anything that can be computed!

Boolean (logical) expressions

These expressions used by the *if* and *while* statements take on the values

- *true (not zero)* and
- *false (zero).*

Simple expressions commonly involve two numeric expressions and a *comparison operator:*

```
if (value != 0)
{
    do something
}
else
{
    do something-else
}

while (counter <  (2 * large – 1))
{
    doing something again
    counter = counter + 1;
}
```

More complex expressions can be made by using *logical operators:*

and - &&
or - ||
not - !

The general forms of such expression are:

(boolean expression1 && boolean-expression2)
true if and only if *both* expressions are true.

(boolean expression1 || boolean-expression2)
true if and only if *at least one* of the expressions is true

!boolean-expression
true if and only if the expression is false.

Examples:

Compute the smallest of three variables *a, b,* and *c* known to be different.

```
if ((a < b) && (a < c ))
    min = a;
else if ((b < a) && (b < c ))
   min = b;
else
   min = c;
```

Determine if *val* is between -9 and +9

You might be tempted to try:

```
if (-9 <= val <= 9)  <--- don't do it!!
```

This will not work at all. In languages with Boolean constants and variables it will cause a compile time error. In C the expression is true 100% of the time regardless of the value of *val.*

Instead we must use a compound condition:

```
if ((-9 <= val) && (val <= 9))
```

Determine if *val* is *not* between -9 and +9

```
if (!((-9 <= val) && (val <= 9)))
```

what happens if we forget the ( )

or

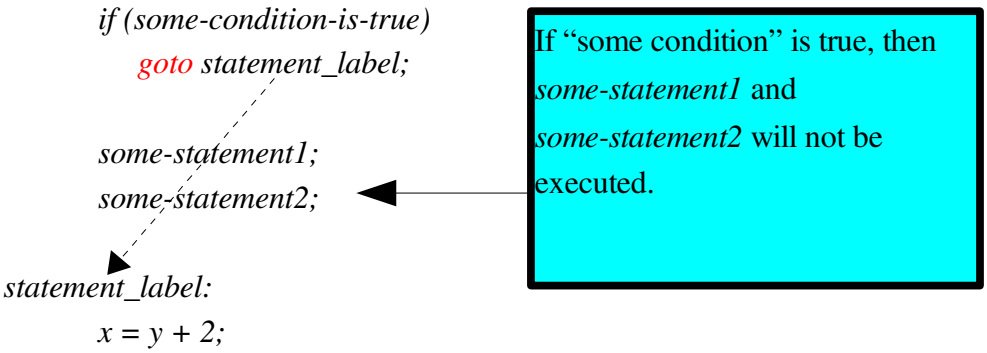
```
if (!(-9 <= val) || !(val <= 9))
```

which can be written as

```
if ((-9 > val)   || (val > 9))
```

Managing flow of control in high level languages

In the early high level languages the *jumpc* instruction was realized as:



Like the *jumpc* instruction the *if with goto* contruct permits us to compute anything that can be computed and this construct is supported in C programs.

Some contend that the use *if with goto* makes programs

- difficult to write correctly and to debug
- difficult for another person to read and understand and therefore
- difficult to maintain

Others contend that a *well-designed program* that uses *if with goto* is no more difficult to understand that one that uses other mechanisms.

Still others contend that using other mechanisms *inherently facilitates good design*.

There is doubtless *some* truth in all of these contentions..

Flow control statements

The while statement

while (expression)
statement

while the value of expression is not zero, repeatedly execute *statement*

while (expression)
basic-block

while the value of expression is not zero, repeatedly execute the entire *basic block.*

It should be clear that the *statement* or the *basic-block following the while should have the ability to eventually make the expression have the value FALSE (0) !!!* If such is not the case the resulting behavior is called an infinite loop. The program will never terminate.

```
int sum;
int val;

val = 0;
while (val <= 100)
{
    sum = sum + val;
    val = val + 1;
}
```

Initialize control variable

Usee control variable in boolean expression

Modify control variable

In the above example when *val is equal to 101, the value of the expression (val <= 100) is 0.*

Uninitialized variables

Use of uninitialized variables is a common error made by both novice and experienced programmers. This error is particularly dangerous because it can create a program that appears to work correctly.

The program fragment shown on the previous page is actually broken, because *sum was not initialized*.

```
int sum;
int val;

val = 0;
while (val <= 100)
{
    sum = sum + val;
    val = val + 1;
}
```

Initialize control variable

Usee control variable in boolean expression

Modify control variable

Preventing uninitialized variables

A 100% effective way to prevent the use of uninitialized variables is to *initialize every variable when it is declared.*

```
int sum = 0;
int val = 0;

while (val <= 100)
{
     sum = sum + val;
     val = val + 1;
}
```

The if statement

if (expression)
    statement or basic-block
else
    statement or basic-block

- If the *expression* is *true (non-zero)* then the statement or basic block following the *if* will be executed.

- If the *expression* is *false (zero)* then the statement or basic block following the *else* will be executed.

```
if (fscanf(stdin, "%d", &number) == 1)
{
    sum = sum + number;
}
else
{
    fprintf(stderr, "Failed to read number\n");
}
```

The complete program

```
acad/cs101/notes06 ==> cat p4.c
#include <stdio.h>

int main()
{
   int howmany = 0;    // how many integers were read
   int num1 = 0;       // the first number of the pair
   int num2 = 0;       // the second number of the pair
   int sum  = 0;

   howmany = fscanf(stdin, "%d %d", &num1, &num2);

   while (howmany == 2)
   {
      sum = num1 + num2;
      fprintf(stdout, "%d  +  %d =  %d \n",
           num1, num2,  sum);
      howmany = fscanf(stdin, "%d %d", &num1, &num2);
   }

   return(0);
}
```

```
acad/cs101/notes06 ==> gcc -g -Wall p4.c -o p4
acad/cs101/notes06 ==> ./p4

1 2
1  +  2 =  3
9 -4
9  +  -4 =  5
-8 3
-8  +  3 =  -5
4 4
4  +  4 =  8
```

Additional output format codes

Format codes can specify how you want to see the integer represented.

%c	Consider the integer to be the ASCII encoding of a character and render that character
%d	Produce the ASCII encoding of the integer expressed as a decimal number
%x	Produce the ASCII encoding of the integer expressed as a hexadecimal number
%o	Produce the ASCII encoding of the integer expressed as an octal number

Specifying field width

Format codes may be preceded by an optional field width specifier. The code ***%02x*** shown below forces the field to be padded with leading 0's if necessary to generate the specified field width.

```
#include <stdio.h>

int main(
int argc,
char *argv[])
{
   int x = 0;
   int y = 78;

   x = 'A' + 65 + 0101 + 0x41 + '\n';
   fprintf(stdout, "X = %d \n", x);

   fprintf(stdout, "Y = %c %3d %02x %4o \n", y, y, y, y);
}

/home/westall ==> gcc -o p1 p1.c
/home/westall ==> p1
X = 270
Y = N  78 4e 116
```

As before the number of values printed by *fprintf()* is determined by the number of distinct format codes.

Applications of Control Flow Mechanisms

In this section we will examine some basic algorithms that employ the *while* and *if* control mechanisms. Important applications include combinations of:

- *Counting things*
- *Accumulating (summing) totals*
- *Searching for specific values*
- *Recurrences*

Examples will be based upon a common design:

Initialize program state and read the first value
While (the number of values read is satisfactory)
    update program state as needed
    read next value(s)
Output final state

The type of state that must be maintained by the program that must be maintained by the program is dependent on the nature of the problem and can include:

- *indicator (true/false) variables*
- *counter variables*
- *accumulator (sum) variables*
- *previous input value(s)*

Counting: Find the number of values in a file of integers

Here the program state that must be maintained is a counter that maintains the number of values that have read so far. As each new value is obtained, the counter is incrmented by one. A common mistake is to *forget to initialize the value of a counter or accumulator variable!* This mistake is particularly evil because it can produce a program that sometimes works and sometimes does not depending upon whether the last value to occupy the memory used the the counter or accumulator was a 0. Therefore we use the technique proposed earlier to prevent this.

```
/* p6.c */
#include <stdio.h>

int main()
{
   int counter = 0;    // the number of values read
   int value = 0;      // the value just read
   int howmany = 0;    // howmany values were read

   counter = 0;
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      counter = counter + 1;
      howmany = fscanf(stdin, "%d", &value);
   }

   fprintf(stdout, "The number of values was %d \n",
                        counter);
   return(0);
}
```

```
Sample input: -11 12 -1 15
Sample output: The number of values was 4

Sample input: 14 12 9
Sample output: The number of values was 3
```

Conditional Counting: Find the number of 3's in an input file of integer values

In this program the state that must be maintained is the number of 3's seen so far. Therefore, it is necessary to condition the incrementing of counter upon whether or not the value just read was a 3. The *if ()* statement is designed for precisely this purpose.

```
/* p5.c */
#include <stdio.h>

int main()
{
   int counter = 0;    // the number of three's we've seen
   int value = 0;      // the value just read
   int howmany = 0;    // howmany values were read

1  counter = 0;
2  howmany = fscanf(stdin, "%d", &value);

3  while (howmany == 1)
   {
4     if (value == 3)
5         counter = counter + 1;

6    howmany = fscanf(stdin, "%d", &value);
   }
7  fprintf(stdout, "The number of 3's was %d \n",
                      counter);
8  return(0);
}
```

```
Sample input: 1 2 3 2 1 3 4
Sample output: The number of 3's was 2

Sample input: 3 3 3
Sample output: The number of 3's was 3
```

Exercise: Build a trace table for the execution of this program given an input of
3 0 3

Accumulation: Find the sum of all of the numbers in a file

Here the program state that must be maintained is the *sum of all values that have seen so far*. As each new value is obtained, its value is added to the current value of *sum*. A common mistake is to forget to initialize the value of an accumulator variable!

```
/* p6.c */
#include <stdio.h>

int main()
{
   int sum = 0;     // the sum of the single digit numbers
   int value = 0;   // the value just read
   int howmany = 0; // howmany values were read

1   sum = 0;
2   howmany = fscanf(stdin, "%d", &value);

3   while (howmany == 1)
   {
4     sum = sum + value;
5     howmany = fscanf(stdin, "%d", &value);
   }

6  fprintf(stdout, "The sum was %d \n", sum);
7  return(0);
}
```

```
Sample input: -11 12 -1 15
Sample output: The sum was 15

Sample input: 14 12 9
Sample output: The sum was 35
```

Exercise: Build a trace table for the execution of this program for the input
-1 3 -2

Conditional Accumulation: Find the sum of all of the positive numbers in a file

Here the program state that must be maintained is the sum of all the positive values that have seen so far. As each new value is obtained, *if the value is positive,* it is added to the current value of *sum* .

```
/* p6.c */
#include <stdio.h>

int main()
{
   int sum = 0;     // the sum of the positive numbers
   int value = 0;   // the value just read
   int howmany = 0; // howmany values were read

   sum = 0;
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      if (value > 0)
         sum = sum + value;
      howmany = fscanf(stdin, "%d", &value);
   }

   fprintf(stdout, "The sum was %d \n", sum);
   return(0);
}
```

```
Sample input: -11 12 -1 15
Sample output: The sum was 27

Sample input: 14 12 9
Sample output: The sum was 35
```

Conditional Accumulation: Find the sum of all of the single digit numbers in a file

The first problem we have to address is *what is a single digit number.* We might initially think of 0, 1, 2, .. 9, but we would be forgetting -9, -8,.. -1!! So the program state that must be maintained is the sum of all values between [-9 and 9] that have seen so far. As each new value is obtained, it must be tested to see if it is a single digit and if so its value is added to the current value of *sum*.

```
/* p6.c */
#include <stdio.h>

int main()
{
   int sum = 0;     // the sum of the single digit numbers
   int value = 0;   // the value just read
   int howmany = 0; // howmany values were read

   sum = 0;
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      if ((value > -10) && (value < 10))
         sum = sum + value;
      howmany = fscanf(stdin, "%d", &value);
   }

   fprintf(stdout, "The sum was %d \n", sum);
   return(0);
}
```

Parentheses should ALWAYS be used in compound conditions to ensure proper evaluation order

```
Sample input: -11 -12 -1 15 2 7
Sample output: The sum was 8

Sample input: 14 12 99
Sample output: The sum was 0
```

Conditional Accumulation: Find the sum of all of the multi-digit numbers in a file.

The first problem we have to address is *what is a multidigit digit number.* The easiest way to do this is to think that its is a number that is *not* a single digit number!! Thus we simply insert the not operator and another set of parentheses in the if expression. We could also use the or operator write the if expression as:

```
if ((value <= -10) || (value >= 10))
if (!((value > -10) && (value < 10)))
```

These two statements are equivalent.

```
/* p6.c */
#include <stdio.h>

int main()
{
   int sum = 0;     // the sum of the multi-digit numbers
   int value = 0;   // the value just read
   int howmany = 0; // howmany values were read

   sum = 0;
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      if (!((value > -10) && (value < 10)))
         sum = sum + value;
      howmany = fscanf(stdin, "%d", &value);
   }

   fprintf(stdout, "The sum was %d \n", sum);
   return(0);
}
```

```
Sample input: -11 -12 -1 5 2 7
Sample output: The sum was -24

Sample input: 1 2 -9
Sample output: The sum was 0
```

Searching: Write a program that reads a collection of integers from the standard input. If the collection contains the value 13 print "yes" to the standard output. Otherwise print "no".

The state that must be maintained here is an indicator *(true/false)* value that defines whether or not the program has encountered a value of 13 in its input. We use the standard C language convention of representing *false* with a value of 0 and *true* with a value of 1.

```
/* p9.c */
#include <stdio.h>

int main()
{
   int found13 = 0; // indicator var: 1 => found a 13
   int value = 0;   // the value just read
   int howmany = 0; // howmany values were read

   found13 = 0;     // haven't found it yet
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      if (value == 13)
         found13 = 1;
      howmany = fscanf(stdin, "%d", &value);
   }

   if (found13 == 1)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);
}
```

We could also say `if (found13)` here.

Exercise: What would the output be if this if/else statement were included in the while loop??

```
Sample input:
1 2 4 13 77 12 1
Sample output:
yes
```

Note: This program could be improved by having it print "yes" and terminate as soon as it finds the first 13, but the program as written is correct. In this course our focus will be building correct programs that might run more slowly than optimal... not incorrect ones that run really fast.

Searching: Find and print the smallest value in a file containing a collection of integers.

This problem is somewhat more subtle than determining whether or not the file contains a "13".

Here the state that must be maintained is *the smallest value seen so far.* It is tempting to initialize this to a large number. However, that approach is not a good idea. The proper way to handle problems of this sort is to initialize the state variable minval to the *first value in the file.*

```
/* p11.c */
#include <stdio.h>

int main()
{
   int howmany = 0;     // how many integers were read
   int minval = 0;      // the minimum value seen so far
   int value = 0;       // current value

   howmany = fscanf(stdin, "%d", &value);
   minval = value;

   while (howmany == 1)
   {
      if (minval > value)
         minval = value;

      howmany = fscanf(stdin, "%d", &value);
   }

   fprintf(stdout, "%d\n", minval);
   return(0);
}
```

```
Sample input: 4 -100 5
Sample output: -100

Sample input: 10000 100000 200000
Sample output: 10000
```

Searching: Write a program that reads a collection of integers from the standard input. If the collection contains a value of 13 followed by a 14 print "yes" to the standard output. Otherwise print "no".

The state that must be maintained by this program is more complex than that of the previous one. Here an indicator of whether the subsequence {13, 14} has or has not been seen yet must be maintained. It is *not possible to determine if such is the case by looking at only the current input value.* If the current input value is 14, the program must also know whether the previous input value was 13. Therefore, it is necessary to always remember the previous value. The variable *old* is used for that purpose. Initialization is also more complicated.

```
/* p10.c */
#include <stdio.h>

int main()
{
   int old = 0;        // previous value
   int value = 0;      // current value
   int howmany = 0;    // howmany values were read
   int found = 0;      // found {13, 14}

   howmany = fscanf(stdin, "%d", &old);
   howmany = fscanf(stdin, "%d", &value);
```

What happens here if the input file contains only 0 or 1 values? Should *howmany* be tested after each call to *fscanf()*??

The main loop illustrates two important programming techniques:

- the *if* statement with compound condtions (always use parentheses)
- a *window mechanism* for discarding the oldest value and updating older, old, and value.

```
   while (howmany == 1)
   {
      if ((old   == 13) &&
          (value == 14))
      {
         found = 1;
      }

      old = value;
      howmany = fscanf(stdin, "%d", &value);
   }

   if (found)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);
}
```

The order in which these assignments are made is critical to correctness.

```
Sample input: 11 12 13 15 12
no
Sample input: 14
no
```

Exercise: Modify the program so that it prints yes if and only if the file contains at least one 13 at least one 14 in ANY order.

Exercise: Modify the program so that it prints yes if and only if the file contains one or more 13's or 14's (This is a good midterm exam type problem).

Exercise: Modify the program so that it prints yes if and only if the file contains exactly one 13,and exactly one 14

Searching: Write a program that reads a collection of integers from the standard input. If the collection contains a value of 13 followed by a 14 and then by 15 print "yes" to the standard output. Otherwise print "no".

The state that must be maintained by this program is more complex than that of the previous one. Here an indicator of whether the subsequence {13, 14, 15} has or has not been seen yet must be maintained. However, it is clearly *not possible to determine if such is the case by looking at only the current input value and the previous value.* If the current input value is 15, the program must also know whether the previous input value was 14 and the one before that was 13. Therefore, it is necessary to always remember the two previous values. The variables old and older are used for that purpose. Initialization is also more complicated as all three value holders must be initialized.

```
/* p10.c */
#include <stdio.h>

int main()
{

   int older = 0;     // value before that
   int old = 0;       // previous value
   int value = 0;     // current value
   int howmany = 0;   // howmany values were read
   int found = 0;     // found target

   howmany = fscanf(stdin, "%d", &older);
   howmany = fscanf(stdin, "%d", &old);
   howmany = fscanf(stdin, "%d", &value);
```

What happens here if the input file contains only 1 or 2 values? Should howmany be tested after each call to *fscanf()*??

The main loop illustrates two important programming techniques:

- the *if* statement with compound condtions (always use parentheses)
- a *window mechanism* for discarding the oldest value and updating older, old, and value.

```
   while (howmany == 1)
   {
      if ((older == 13) &&
          (old   == 14) &&
          (value == 15))
      {
         found = 1;
      }

      older = old;
      old = value;
      howmany = fscanf(stdin, "%d", &value);
   }

   if (found)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);
}
```

The order in which these assignments are made is critical to correctness.

```
Sample input: 11 12 13 15 12
no
Sample input: 14
no
```

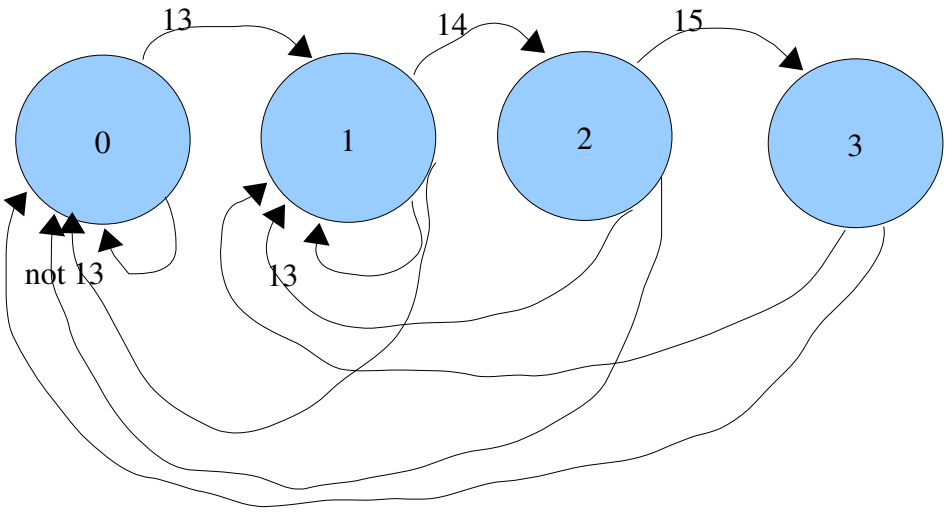
Exercise: Modify the program so that it prints yes if and only if the file contains at least one 13 at least one 14 and at least one 15 in ANY order.

Exercise: Modify the program so that it prints yes if and only if the file contains one or more 13's,14's or 15's. (This is a good midterm exam type problem).

Exercise: Modify the program so that it prints yes if and only if the file contains exactly one 13, exactly 14 and, and exactly one 15 in ANY order.

A state machine approach

Many problems that involve searching for complex subsets can be solved using a state machine. The state machine exists in a finite number of distinct states.



For each state and possible input value a rule specifies the new state. For each state it is necessary that a rule specify the next state for *all possible inputs.* For mapping the table to code its best to put the specific inputs first.

Existing state	*Input value*	*New state*
0	13	1
0	other	0
1	13	1
1	14	2
1	other	0
2	14	3
2	13	1
2	other	0
3	any	3

The rule table can map directly to an *if* structure

```
if ((state == 0) && (value == 13))
    state = 1;
else if (state == 0)
    state = 0;
else if ((state == 1) && (value == 13))
    state = 1;
else if ((state == 1) && (value == 14))
    state = 2;
    :
    :
    :
```

Some clever mental manipulation can greatly reduce the number of tests! But oversimplification can be a threat to correctness!!

A state machine based version of the program

```
#include <stdio.h>

/* We can also use a state machine model instead of */
/* remembering old values                           */
/* state            meaning                         */
/*   0            looking for 13                    */
/*   1            found 13 looking for 14           */
/*   2            found 14 looking for 15           */
/*   3            found 13, 14, 15                  */

int main()
{
   int state;
   int value;          // current value
   int howmany;        // howmany values were read
   int found = 0;      // found target

   howmany = fscanf(stdin, "%d", &value);

   while ((howmany == 1) && (state != 3))
   {
      if (value == 13)
         state = 1;
      else if ((state == 1) && (value == 14))
         state = 2;
      else if ((state == 2) && (value == 15))
         state = 3;
      else
         state = 0;

      howmany = fscanf(stdin, "%d", &value);
   }

   if (state == 3)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);

}
```

Programs with no input

The last two examples that we consider in this section are those in which the program has no input at all! In both cases the set of all non-negative integers is implicitly the input and the problem that we are attempting to solve is to find a subset of the integers that have a particular property.

Searching and enumerating: Print all of the integers less than or equal to 10000 that are perfect squares.

This is a problem in which it pays to engage the brain before engaging the fingers. The first instinct of most people is to try to take the square root of all integers between 1 and 10000 and see if the value is an integer. However, a better approach is to just compute them all directly. In any problem of *enumeration* (print all of the numbers that....) it will be necessary to have *fprintf()* within the main loop (possibly guarded by an *if*. This is in contrast to previous problems in which we printed only the final state of the program at the end.

```
/* p8.c */
#include <stdio.h>

int main()
{
   int val = 1;       // the current value in [1, 10000]
   int valsqr = 0;    // square of the value

   valsqr = val * val;
   while (valsqr <= 10000)
   {
      fprintf(stdout, "%d\n", valsqr);

   /* Compute next value and next square */

      val = val + 1;
      valsqr = val * val;
   }
}
Sample output
1
4
9
16
:
10000
```

Recurrences: Suppose the first two numbers of a sequence of numbers are {0, 1}. Suppose each subseqent number is the sum of its two immediate predecessors. We manually compute a few terms of the sequence as follows:

0 + 1 = 1 -> {0, 1, 1}
1 + 1 = 2 -> {0, 1, 1, 2}
1 + 2 = 3 -> {0, 1, 1, 2, 3}
2 + 3 = 5 -> {0, 1, 1, 2, 3, 5}
3 + 5 = 8 -> {0, 1, 1, 2, 3, 5, 8}

Problem: Print the first twenty terms of this sequence

Here we need to *recycle t*he strategy from the {13, 14, 15} problem in which we learned how to "remember" a window of three values.

```
/* p7.c */
#include <stdio.h>

int main()
{
   int old;     // most recent old value;
   int older;   // less recent old value;
   int new;     // new value;
   int counter; // number of values printed so far

   older = 0;
   fprintf(stdout, "%d\n", older);

   old = 1;
   fprintf(stdout, "%d\n", old);

   counter = 2;
   while (counter < 20)
   {
      new = old + older;
      fprintf(stdout, "%d\n", new);
      counter = counter + 1;
      older = old;
      old = new;
   }
}
```

Why not use
`while (counter <= 20)`
here

Exercises

1. Write a program that reads one integer value at a time from the standard input. If the collection of input integers contains at least one 7 and at least one 11, the program should write "yes" (without the "s) to the standard output. Otherwise it should write "no" to the standard output.

```
7 7 4 5 3
no
7 7 4 2 11 24
yes
```

2. Write a program that reads one integer value at a time from the standard input. If the collection of input integers contains a 7 followed immediately by an 11, the program should write "yes" (without the "s) to the standard output. Otherwise it should write "no" to the standard output.

```
7 7 4 2 11 24
no
7 7 4 7 11 24
yes
```

3. Write a program that reads PAIRS of integers from the standard input. If the sum of the two integers in a pair is 10 then the program should print the pair to the standard output.

```
4 7
4 6
4 7
4 6
```

4. Write a program that reads PAIRS of integers from the standard input. The program should print the number of pairs whose sum is 20 to the standard output

```
21 -1
20  1
1
```

5. Write a program that reads PAIRS of integers from the standard input. The program should compute sum of each pair and print the largest sum found to the standard output.

```
2 3
2 6
2 2
2 -5
8
```

Use of the Command Line

Output redirection

The *fprintf(stdout,)* function sends its output to a logical file commonly known as the standard output or simply *stdout.*

When a program is run in the Unix environment, the logical file *stdout* is by default associated with the screen being viewed by the person who started the program.

The > operator may be used on the command line to cause the standard output to be redirected to a disk resident file:

```
acad/cs101/examples/notes ==> p8 > squares.txt
```

A file created in this way may be subsequently viewed using the cat command (or edited using a text editor).

```
acad/cs101/examples ==> cat squares.txt | more

1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
```

Input redirection

In the Unix environment the standard input or *stdin* is by default bound to the keyboard of the person who runs a program. Like the *stdout* the *stdin* may also be redirected to a file. When the *stdin* is redirected, the program reads its input from the disk file to which the *stdin* has been redirected. Tl redirect both stdin and stdout use:

```
p4 <  p4in.txt  > p4out.txt
```

when invoked in this manner when the program *a.out* reads from the stdin via *scanf()* or *fscanf(stdin,.)* the input will actually be read from a file named *p4in.txt* and any data written to the *stdout* will end up in the file *p4out.txt*.

```
acad/cs101/examples/notes ==> cat p4in.txt
3 4
5 6
7 8
9 11
```

```
acad/cs101/examples/notes ==> p4 < p4in.txt > p4out.txt
```

```
acad/cs101/examples/notes ==> cat p4out.txt
3  +  4 =  7
5  +  6 =  11
7  +  8 =  15
9  +  11 =  20
```

Other control mechanisms

The *for* loop

for (init-expression; continue-condition; update-expression)
{
loop-body
}

The *init-expression* is executed one time
The *continue-condition* is evaluated each iteration of the loop *before* the *loop-body* is executed.
The *update-expression* is executed after the *loop-body* is executed.
The *loop-body* is executed if and only if the *continue* condition is true.

```
#include <stdio.h>

int main()
{
   int i;

   for (i = 5; i < 5; i = i + 1)
      printf("in body with i = %d \n", i);

   printf("at end with i = %d \n", i);

   for (i = 3; i < 5; i = i + 1)
      printf("in body with i = %d \n", i);

   printf("at end with i = %d \n", i);
}
```

```
acad/cs101/examples/notes ==> a.out
at end with i = 5

in body with i = 3
in body with i = 4
at end with i = 5
```

The *do ... while()* loop

do
{
loop-body;
} while (continue-condition);

This structure is similar the the *while ()* loop, but unlike, the while loop,

- the *loop-body* will always be executed at least one time.
- a semicolon *must* follow *while (continue-condition);*

```
#include <stdio.h>

int main()
{
   int i = 15;

   do
   {
      printf("in body with i = %d \n", i);
      i  = i + 1;
   }  while (i < 10);
}
```

```
acad/cs101/examples/notes ==> a.out
in body with i = 15
```

Altering control flow within a loop

Two single word statements may be used to alter control flow within a loop:

The *break* statement

- exit the loop containing the break *immediately*
- execution continues at the line immediately following the break

#include <stdio.h>

```
int main()
{
   int i;
   int j;

   for (i = 0; i < 3; i = i + 1)
   {
      for (j = 0; j < 3; j = j + 1)
      {
         printf("in body with (i, j) = (%d, %d)\n",
                                                i, j);
         if (i == j)
            break;
      }
   }
}
```

```
acad/cs101/examples/notes ==> a.out
in body with (i, j) = (0, 0)
in body with (i, j) = (1, 0)
in body with (i, j) = (1, 1)
in body with (i, j) = (2, 0)
in body with (i, j) = (2, 1)
in body with (i, j) = (2, 2)
```

Use of *break* is discouraged as proper design can almost always produce an equally simple implementation with requiring *break*.

Here is a an example of an equivalent program that does not contain the break;

```
int main()
{
   int i;
   int j;

   for (i = 0; i < 3; i = i + 1)
   {
      for (j = 0; j <= i; j = j + 1)
      {
         printf("in body with (i, j) = (%d, %d)\n",
                                                i, j);

      }
   }
}
```

The *continue* statement

- causes a transfer of control to the end of the loop bypassing the remainder of the statements in the body of the loop.

```
#include <stdio.h>
#include <math.h>

int main()
{
   float val;
   float sqrtval;
   float sum;

   while (fscanf(stdin, "%f", &val) == 1)
   {
      printf("\n%8.2f ", val);

      if (val < 0)
         continue;

      sqrtval = sqrt(val);
      sum = sum + sqrtval;
      printf(" %8.2f %8.2f ", sqrtval, sum);
   }
}
```

```
    1.00      1.00     1.00
    2.00      1.41     2.41
    4.00      2.00     4.41
   -4.00
    5.00      2.24     6.65
```

As with the *break* statement, it is almost always the case that use of the *continue* statement can always be avoided by careful design.

The *switch* statement

Similar in function to the *if , else if, else if* construct.

```
switch (expression)
{
        case const1:
                statement1;
                statement2;
                break;
        case const2:
                statement3;
                break;
        default:
                statement4;
}
```

This is equivalent to:

```
if (expression == const1)
{
        statement1;
        statement2:
}
else if (expression == const2)
{
        statement3;
}
else
{
        statement4;
}
```

Differences

with if/else *const1 and const2* may be *expr1* and *expr2*
the break statement may be removed in the switch to allow *fall-through.*

Example: Counting area code instances

```
#include <stdio.h>

int main()
{
   int areacode;
   int sum864 = 0;
   int sum803 = 0;
   int other  = 0;

   while (scanf("%d", &areacode) == 1)
   {
      switch (area)
      {
      case 803:
         sum803 += 1;
         break;
      case 864:
         sum864 += 1;
         break;
      default:
         other += 1;
      }
   }
}
```

The *question mark* operator

condition ? true-expression: false-expression

The condition is evaluated and if true *true-expression* will be executed and the value of the entire expression will be the value of the *true-expression.*
Otherwise the *false-expression will be executed* and the value of the entire expression will be the value of the *false-expression.*

```
#include <stdio.h>
#include <math.h>

int main()
{
   float val;
   float sqrtval;
   float sum;

   while (fscanf(stdin, "%f", &val) == 1)
   {
      printf("\n%8.2f %8.2f ",
                  val, val > 0 ? sqrt(val): 0.0);

   }
}
```

```
    1.00     1.00
    2.00     1.41
    4.00     2.00
   -4.00     0.00
    5.00     2.24
```

The *comma* operator

e1, e2

This operator can be used when it is desired to place two (or more) expressions where one is normally used. The value of the expression is the value of *e2*.

```
    for (i = 0, j = n - 1; i < n; i++, j--)
    {
        out[j] = in[i];
    }
```

```
#include <stdio.h>

int in[10];
int out[10];

int main()
{
   int i, j;
   int k;
   int m;

   for (i = 0, j = 9; i < 10; m++, k++, i++, j--)
      out[j] = in[i];
}
```

Functions

Functions are the building blocks of C programs.

A program that is more than 25 lines long should be comprised of separate functions that are not longer than 25 lines! Advantages of the use of functions include:

- Facilitates *top down design*
- Facilitates *bottom up implementation and testing*
- *Improves program readability* ... proper function naming can make a program nearly self-documenting.
- *Reduction of nesting*
- Facilitates *reuse of code*

From the Linux kernel Coding Style Guide
Chapter 5: Functions

Functions should be short and sweet, and do just one thing. *They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24), and do one thing and do that well.*

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

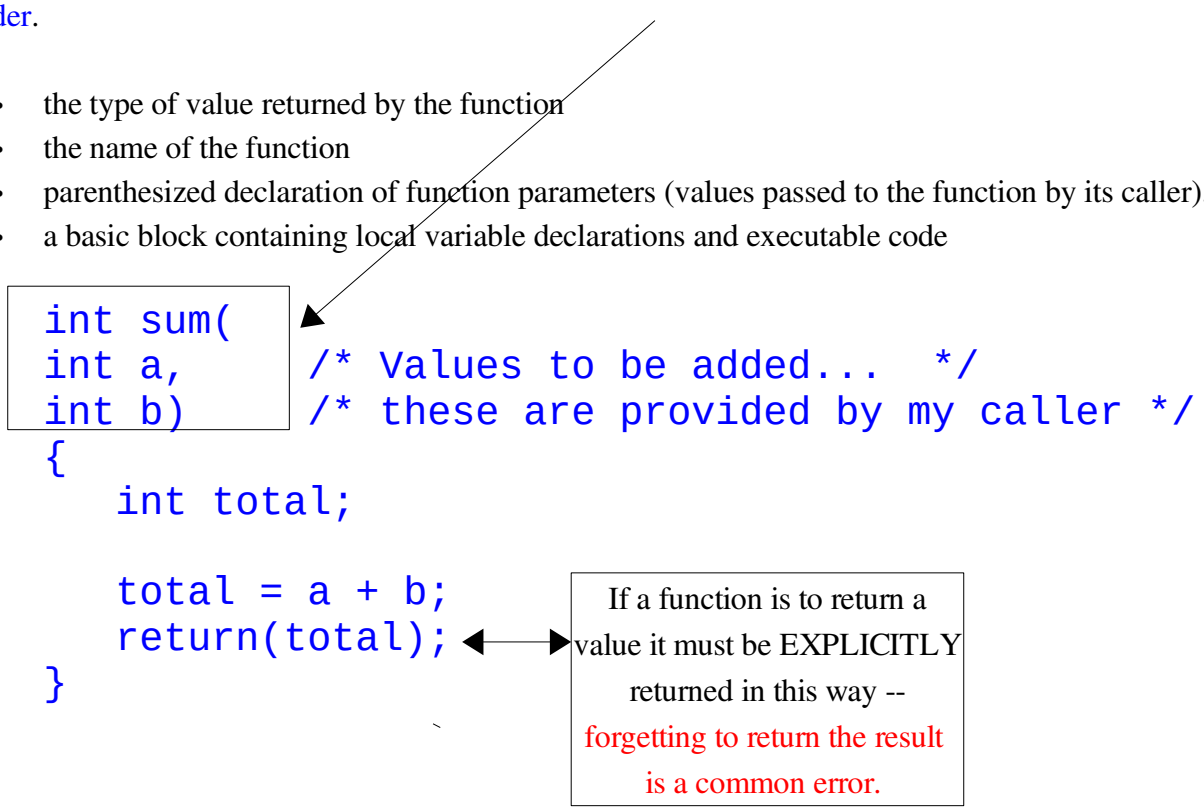
However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

Function definitions

A C function is comprised of 4 components. The first three are collectively referred to as the function header.

- the type of value returned by the function
- the name of the function
- parenthesized declaration of function parameters (values passed to the function by its caller).
- a basic block containing local variable declarations and executable code

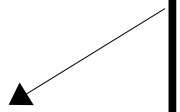


Invoking a function

A C function is called or invoked by just using its name anywhere a variable or constant might be used in an expression:

```
int main()
{
    int num = 0;

    num = sum(4, 5);
    num = num + sum(num, 6);
}
```



Actual arguments passed to the function may be constants or variables. Parameter names used within the function correspond *positionally* to the actual arguments!

When an expression is evaluated, the value returned by the function replaces its invocation in the expression. So here, the first call to sum() will return a 9 and 9 will be assigned to num. Then the second call will return a 15 and so the final value of num will be 15.

NOTES:

- Local variable name spaces of different functions are completely disjoint. I can declare a local variable called total in main() and it will be completely independent of the total in sum.

- The correspondence between the formal parameters (a, and b) in sum() with the actual arguments (4, and 5) in main() is positional. When this program runs, if we were to include

  ```
  fprintf(stdout, "%d %d\n", a,b);
  ```

  in sum(), we would see a holds the value 4 and b holds 5 during the first call and a holds the value 9 and b holds 6 during the second call.

- Therefore, the number and types of the actual arguments used when invoking a function should exactly match the formal parameters of the function definition.

Parameter passing

There are many possible mechanisms by which parameters may be passed.

The standard C language uses *pass-by-value* for passing all scalar data types.

In this approach a *copy* of the parameter is placed on the stack. The called function is free to modify the copy as it wishes, but this will have *no effect* on the value held by the caller.

```
/* Parameter passing 1 */

int try_to_mod(
int a)
{
   printf("The address of try's a is %p\n", &a);
   a = 15;
}

int main()
{
   int a = 20;

   printf("The address of main's a is %p\n", &a);
   try_to_mod(a);
   printf("a = %d \n", a);
   return(0);
}
```

```
class/215/examples ==> a.out
The address of main's a is 0xbffff884
The address of try's a is 0xbffff870
a = 20
```

Note that the two variables named *a* occupy different memory locations.

The order of functions in a C program:

The C language generally requires that all entites be defined in a program before they are referenced.

```
acad/cs101/examples/notes ==> cat p24.c
int main()
{

   x = sum(5, 3);
   int x;
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}
```

```
acad/cs101/examples/notes ==> gcc -Wall p25.c
p25.c: In function ‘main’:
p25.c:5: error: ‘x’ undeclared (first use in this function)
p25.c:5: error: (Each undeclared identifier is reported only once
p25.c:5: error: for each function it appears in.)
p25.c:5: warning: implicit declaration of function ‘sum’
p25.c:6: warning: unused variable ‘x’
```

For functions the rule may appear somewhat relaxed.

The compiler may produce a warning instead of an error and proceed to compile the program.

Many people believe this relaxation is a *bad idea.*

Here we fix the declaration of *x*

```
acad/cs101/examples/notes ==>

int main()
{
   int x;

   x = sum(5, 3);
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}
```

The compilation works but produces a *nastygram.*

```
acad/cs101/examples/notes ==> gcc -Wall p24.c
p24.c: In function ‘main’:
p24.c:6: warning: implicit declaration of function ‘sum’
acad/cs101/examples/notes ==>
```

Other times the compiler may generate an error.

The error occurs because the *default return type* for an undeclared function is *int.*

- *At line 7 sum was implicitly assumed to return int.*
- *At line 15 sum was seen to actually return unsigned char*

```
#include <stdio.h>

int main()
{
   int x;

   x = sum(11, 3);
   fprintf(stdout, "%d \n", x);
   return(0);
}

unsigned char sum(
int a,
int b)
{
   return(a + b);
}
```

Line 7

Line 15

```
acad/cs101/examples/notes ==> gcc -Wall p29.c
p29.c: In function ‘main’:
p29.c:7: warning: implicit declaration of function ‘sum’
p29.c: At top level:
p29.c:15: error: conflicting types for ‘sum’
p29.c:7: error: previous implicit declaration of ‘sum’ was
here
```

Consequences of ignoring the warning

Even though only a warning may issued by the compiler, you may get a defective result when the function is not defined at the time it is called.

Here the compiler can't know what type of parameters *sum* is expecting. Therefore, it passes it a *floating point* constant 11.0 and *int 3*. The *sum* function assumes that its first parameter is an *int* and big trouble ensues.

```
#include <stdio.h>

int main()
{
   int x;

   x = sum(11.0, 3);
   printf("%d \n", x);
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}
```

```
acad/cs101/examples/notes ==> !gcc -Wall
acad/cs101/examples/notes ==> gcc -Wall p27.c
p27.c: In function ‘main’:
p27.c:7: warning: implicit declaration of function ‘sum’
```

When the program actually runs, the floating point representation of 11.0 is treated as an integer by sum and big trouble ensues.

```
acad/cs101/examples/notes ==> a.out
1076232192
```

Avoiding the warnings and the errors

One way to do this is just to order functions so that the definition always appears before the first invocation of the function.

```
acad/cs101/examples/notes ==> cat p28.c
#include <stdio.h>

int sum(
int a,
int b)
{
   return(a + b);
}

int main()
{
   int x;

   x = sum(11.0, 3);
   printf("%d \n", x);
   return(0);
}

acad/cs101/examples/notes ==> a.out
14
```

Function prototypes

The ordering strategy proposed on the previous page can't solve the problem in two situations:

- The function definition is located in another source module (.c file)
- Two or more functions are mutually referential. *a calls b which calls c which calls a.*

A *prototype* is a function header which is *followed by a semicolon instead of a basic block*. The prototype tells the compiler:

- the type of value returned by the function
- the type of each parameter

When the compiler has this information it will generate code that will automagically coerce actual arguments to the correct type.

An example prototype for our sum function is thus:

```
int sum(int a, int b);
```

Proper use of prototypes

When the prototype is placed at the start of the source module, the placement of the function becomes irrelevant and even self-referential functions work properly.

```
#include <stdio.h>

int sum(int a, int b);

int main()
{
   int x;

   x = sum(11.0, 3);
   printf("%d \n", x);
   return(0);
}

int sum(
int a,
int b)
{
   return(a + b);
}
acad/cs101/examples/notes ==> gcc -Wall p30.c
acad/cs101/examples/notes ==> a.out
14
acad/cs101/examples/notes ==>
```

The prototype

Compiler now generates code to convert the *float* value to *int* before passing it to sum!

Prototypes and .h (header) files

In developing large-scale C program extensive use of both *C Library* functions and *user written* functions is common. Such programs are also comprised of multiple *.c* files with intra-module function calls being common.

Physically including all the prototypes in each source module and keeping them syncrhonized in the event of changes is not feasible. Therefore function header files are used to consolidate prototypes. Such files are commonly called *.h* files and are included into the source file at compile time by means of the *#include* directive.

- Function prototypes for *standard library functions* such as *fscanf()* and *fprintf()* are always available in a *.h* file. Failure to include the proper *.h* file can lead to failure of your program. Use of *< > notation instructs the C compiler to search the standard library /usr/include* for the .h file.

```
#include <stdio.h>
```

- Programmers also create their own .h files containing prototypes for the functions that they write. Use of " " notation instructs the C compiler to look in the current working directory.

```
#include "myhdrs.h"
```

PPM Images

Representations of image data

- Images (e.g. digital photos) consist of a rectangular array of discrete picture elements called *pixels.*
- An image consisting of 200 rows of 300 pixels per row contains 200 x 300 = 60,000 individual pixels.
- The Portable PixMap (.ppm) format is a particularly simple way used to encode a rectangular image (picture) as uncompressed data file.
- The .ppm file can viewed with a number of tools including *xv, display,* and *gimp*.
- Other well known formats include JPEG (.jpg), TIFF (.tif), GIF (.gif), and PNG (.png)

We can use *.ppm* to represent two types of images

Grayscale images –

- correspond to black / white photographs
- each pixel consists of *1 byte* which should be represented as unsigned char
- a value of 0 is solid black
- a value of 255 is bright white
- intermediate are "gray" values of increasing brightness.

Color images -

- correspond to color photographs
- each pixel consists of *3 bytes* with each byte represented by an unsigned char
- this format is call RGB three bytes represent the
    - red component
    - green component
    - blue component
- When red == green == blue a grayscale "color" is produced. Thus, a grayscale picture can be stored (somewhat inefficiently) in a color ppm file --- but a color image can *never* be stored in a grayscale file.
- (255, 255, 255) is bright white

Colors are additive

- (255, 255, 0) = red + green = bright yellow
- (255, 0, 255) = red + blue = magenta (purple)
- (0, 255, 255) = blue + green = cyan (turquoise)
- (255, 255, 255) = red + green + blue = white

The Irish national flag:

It can be quite challenging to figure out the precise (R, G, B) intensities that are needed to produce a particular color.

A Wikipedia lookup can help:

Scheme	Green	White	Orange
RGB	0-154-99	255-255-255	255-130-61
Hex	#009A63	#FFFFFF	#FF823D

PPM file structure

ppm header
packed image data

The .ppm header

```
P6
# This is a comment
# So is this... (X, y) dimensions follow
600 400
# Maximum value of a pixel. Ours will always be 255
255
```

- The P6 indicates this is a color image (P5 means grayscale)
- The *width* of the image is 600 pixels
- The *height* of the image is 400 pixels
- The 255 is the maximum value of a pixel.
- Following the 255 is a \n (0x0a) character.

The image data

- The red component of the upper left pixel must immediatly follow the new line.
- There must be a total of 3 x 600 x 400 = 720,000 bytes of data.

Building a .ppm file

```
/* p12.c */

/* Construct a solid color image of the specified color */

/* Input data                                                 */
/*     width  in pixels  height in pixels                     */
/*     red value  green value  blue value                     */

#include <stdio.h>

void make_ppm_header(int width, int height);
void make_ppm_pixel(unsigned char r, unsigned char g,
                      unsigned char b);
void make_ppm_image(int width, int height, unsigned char r,
                    unsigned char g, unsigned char b);

int main(
{
   int width;
   int height;
   int count = 0;

   unsigned int red;
   unsigned int green;
   unsigned int blue;

/* Read image dimensions and pixel color */

   fscanf(stdin, "%d %d", &width, &height);
   fscanf(stdin, "%d %d %d", &red, &green, &blue);

/* Create the .ppm the image file */

    make_ppm_image(width, height, red, green, blue);

   return(0);
}
```

Writing a .ppm header

The mission of *make_ppm_header()* is to print a proper *.ppm* header to the standard output. A single newline should immediately follow the 255. *Inserting extra space characters produces color shifts in which red becomes green, green becomes blue and blue becomes red.!!*

```
void make_ppm_header(int wd, int ht)
{
   fprintf(stdout, "P6\n");
   fprintf(stdout, "%d %d 255\n", wd, ht);
   return;
}
```

Writing a single pixel

The %c format code must be used to write the pixel value as a single byte without conversion to ASCII text that %d would produce. *Inserting extra space characters here also produces color shifts.*

```
void make_ppm_pixel(unsigned char r, unsigned char g,
unsigned char b)
{
    fprintf(stdout, "%c%c%c", r, g, b);
}
```

%c format prevents data conversion
%c format generates 1 byte of output
no blank spaces permitted in format string

Creating the .ppm image

```
void make_ppm_image(int width, int height, unsigned char r,
                    unsigned char g, unsigned char b)
{
   int count = 0;

/* Write the ppm header */

   make_ppm_header(width, height);

/* Write the ppm data */

   while (count < (width * height))
   {
      make_ppm_pixel( r, g, b);
      count = count + 1;
   }
}
```

Sample input file

```
class/101/examples ==> cat pic1.in
200 150
200  64 224
```

Width and height in *pixels*

Intensity of R, G, and B pixels.

Running the program with input and output redirection

```
p12 < pic1.in > pic1.ppm
```

Viewing the data with a hexadecimal / ASCII dump utility shows that the header comprises the first 15 bytes of the file. The remainder of the file contains the bytes C8 40 E0 repeated 150 x 200 = 30,000 times.

```
od -t x1a pic1.ppm | more

0000000 50 36 0a 32 30 30 20 31 35 30 20 32 35 35 0a c8
         P  6 nl  2  0  0 sp  1  5  0 sp  2  5  5 nl  H
0000020 40 e0 c8 40 e0 c8 40 e0 c8 40 e0 c8 40 e0 c8 40
         @  `  H  @  `  H  @  `  H  @  `  H  @  `  H  @
0000040  etc
```

Note that

200 base 10 is c8 base 16

64 base 10 is 40 base 16 (and is the ASCII code for the @ character)

224 base 10 is E0 base 16

The *ls -l* command command can be used to show the size of the entire file

```
class/101/examples ==> ls -l pic1.ppm
-rw-------  1 westall iiimd 90015 Sep 13 13:03 pic1.ppm
```

15 (header length)

3x 200 x 150 = 90000 (image data)

90015

This is the image:

Building a .ppm file part 2

```
/* p12b.c */

/* Construct a continuously varying color image          */
/* The red component is a function of pixel row          */
/* The blue component is a function of pixel col         */
/* The blue component is 0                               */
/* Input data                                            */
/*     width  in pixels  height in pixels                */
/*     red value  green value  blue value                */

#include <stdio.h>

void make_ppm_header(int width, int height);
void make_ppm_row(int width, int height, int row);
void make_ppm_image(int width, int height);

int main()
{
   int width;
   int height;

/* Read image dimensions and pixel color */

   fscanf(stdin, "%d %d", &width, &height);

/* Write the image data */

    make_ppm_image(width, height);

   return(0);
}
```

Writing a .ppm header

The .ppm header writer function we used before can just be recycled! This is what is meant by facilitating reuse of code!

```
void make_ppm_header(int wd, int ht)
{
   fprintf(stdout, "P6\n");
   fprintf(stdout, "%d %d 255\n", wd, ht);
   return;
}
```

The same is true of the pixel writer

```
void make_ppm_pixel(unsigned char r, unsigned char g,
unsigned char b)
{
    fprintf(stdout, "%c%c%c", r, g, b);
}
```

Since the color depends on the (row, col) location of the pixel, we now build the image one row at a time. The red component gets steadily brighter as we proceed top to bottom in the image and the green component gets brighter as we proceed left to right. Therefore the bottom right of the image should be yellow.

```
void make_ppm_image(int width, int height)
{
   int row = 0;

/* Write the ppm header */

   make_ppm_header(width, height);

/* now build the image  one row at a time */

   while (row < height)
   {
      make_ppm_row(width, height, row);
      row = row + 1;
   }
}

void make_ppm_row(int width, int height, int row)
{
   int col = 0;
   unsigned char r;
   unsigned char g;

   while (col < width)
   {
      r = 255 * row / height;
      g = 255 * col / width;
      make_ppm_pixel(r, g, 0);
      col = col + 1;
   }
}
```

Arrays

Arrays provide a useful way to access a block of adajcent memory cells using

- *a single name* and
- *a numeric index*

An array of 100 integers is declared as follows:

```
int nums[100];
```

This declaration

- reserves 100 *ints* = 4 x 100 *bytes* of continuous memory
- the first *int* in the array is accessed using index 0
  - `nums[0]  = x;`
- the last *int* in the array is accessed using index 99
  - `nums[99] = y;`

The term *word* will sometimes be used as a synonym for *int.*

Note well: The context in which the "subscript" value is used determines what it means:

Here it means reserve me 100 ints:

```
int nums[100];
```

Here it means assign 15 to a specfic element in the array

```
nums[100] = 15;
```

Note that such an assignment is illegal, but it will be performed anyway. This is a very common error in programs that use arrays.

Variables as array indices

The real power of the array is the capability to use a *variable* to access individual cells. What does the following loop do?

```
ndx = 0;
while (ndx < 100)
{
   nums[ndx] = ndx;
   ndx = ndx + 1;
}
```

How about this one?

```
ndx = 0;
nums[0] = 0;
while (ndx < 100)
{
   ndx = ndx + 1;
   nums[ndx] = nums[ndx - 1] + 1;
}
```

When to use an array

Beginning programmers sometimes encounter an irresistable urge to *use arrays where they are not needed. Please resist this urge!*

Reasons *to use* an *array*

- The algorithm being used requires repeated access to a whole collection of values (e.g. sorting).
- Economies of scale in input/output exceed the performance penalty of filling a large array.
- Character arrays are commonly used to store words or sentences.

None of the algorithms that were previously presented for counting, summing, searching and recurrences require an array and no array should be used in problems of these types.

For any problem that requires you to read an "unknown" number of values from standard input, there is no way for you to know in advance how large to make the array!!

Passing arrays as parameters

We saw earlier that the C language uses *pass-by-value* when scalar types such as *ints* are passed as parameters to functions. In this approach

- A *copy of the value* is passed to the function
- The function may modifiy its copy of the parameter
- but the value owned by the caller of the function will *not be changed.*

The standard C language uses *pass-by-address* for passing all array data types.

In this approach the *address of* of the parameter is passed to the function.

- The called function can use this address to access *and modify* the actual array that is owned by the caller.

Array parameter example

The name *b[]* in the *try_to_mod()* function is known as an alias for the name *a[]* in the main() function because both names refer to the same physical memory locations.

```
/* Parameter passing 2 */

int try_to_mod(
int b[])
{
   printf("The address of try's b is %p\n", &b[0]);
   b[0]= 15;
}

int main()
{
   int a[3];

   a[0] = 20;
   a[1] = 30;
   printf("The address of main's a is %p\n", &a[0]);
   try_to_mod(&a[0]);
   printf("a[0] = %d \n", a[0]);
   return(0);
}
```

As seen in a previous example, an array can be passed by simply using its name:

```
try_to_mod(a);
```

```
The address of main's a is 0xbf879624
The address of try's b is  0xbf879624
a[0] = 15
```

Note that the variables named *a* in main() and *b* in try_to_mod() occupy the same memory locations.

Unlike the previous example the *try_to_mod()* function *succeeds* in changing data owned by its caller.

A problem that does require the use of an array

The objective of this example is to read in a collection of values from the standard input and print those that are less than or equal to the average. It is *impossible to know what the average is before all the values have been read.* Therefore it is necessary to "remember" all of the values as they are read in by storing them into adjacent elements in an array.

The solution will consist of three steps.

- Read values into an array and determine the number of values read.
- Process the array computing the average value
- Process the array printing values that are less than or equal to the average

```
/* p14.c */
#include <stdio.h>

int fill_array(int data[]);
int find_average(int data[], int counter);
void print_small(int data[], int counter, int avg);

int main()
{
   int values[100]; // holds up to 100 input values
   int counter;     // the number of values read in
   int average;     // average of the values

   counter = fill_array(values);
   average = find_average(values, counter);
   print_small(values, counter, average);
   return(0);
}
```

Reading an unknown number of integers into an array

```
int fill_array(int data[])
{
  int howmany;     // howmany values were read

/* Initialize and read first integer into array */

   int counter = 0;
   howmany = fscanf(stdin, "%d", &data[0]);
```

We *could* say &data[counter] here

In this loop we accomplish two objectives: (1) the *remaining values are read into the array and (2) the number of values is maintained in counter.*.

```
   while (howmany == 1)
   {
      counter = counter + 1;
      howmany = fscanf(stdin, "%d",
            &data[counter]);
   }
   return(counter);
}
```

The order of the the two statements is *critical to correctness*

Exercise: What will happen if we should change the order of the two statements in the loop?

Computing the average of the elements in the array

This next function shows how to carry out a specific number of iterations of a loop. This procedure requires two variables:

- *counter - the total number of iterations to be made which is equal to the number of elements actually read into the array.*
- *ndx – the current iteration number which takes on the values 0, 1, ... counter-1*

```
int find_average(
int data[],
int counter)
{
   int ndx = 0;
   int sum = 0;
   int average = 0;

/* When the read loop ends the value of counter */
/* is the number of elements in the array and  */
/* so (counter – 1) is the largest valid index */

   while (ndx < counter)
   {
      sum = sum + data[ndx];
      ndx = ndx + 1;
   }
   average = sum / counter;
   return(average);
}
```

Remember to initialize ndx and sum

We can't say data[0] here

Forgetting to increment the index is a very common way to accidentally create an inifinite loop.

Printing the only those elements whose value is <= to the average.

In this function another complete pass is made over the entire array. Needless to say it is critical to remember to set the value of *ndx* to 0 before starting the loop.

```
void print_small(
int data[],
int counter,
int avg)
{
   int ndx = 0;

   while (ndx < counter)
   {
      if (data[ndx] <= avg)
         fprintf(stdout, "%d \n",  data[ndx]);
      ndx = ndx + 1;
   }
}
```

It is absolutely necesssary to initialize *ndx*

Running the program produces the expected output:

```
==> p14
1 10 2 9 3 8 4 7 5 6
1
2
3
4
5
```

A problem with no pleasant solution

What if the number of values in the input file is 110?

- The program as written will over write unallocated memory causing either a program fault or incorrect output!
- One approach could be to just abort the program if *too many* numbers are provided in the input.

```
while (howmany == 1)
{
   counter = counter + 1;
   if (counter == 100)
   {
       fprintf(stderr, “Too many values \n”);
       exit(-1);
   }
   howmany = fscanf(stdin, "%d",
          &data[counter]);
}
```

A call to the *exit()* function terminates the program!

An alternative approach is to process *only* the first 100 values.

```
while ((howmany == 1) && (counter < 99))
{
   counter = counter + 1;
   howmany = fscanf(stdin, "%d", &data[counter]);
}
```

But obviously what we would really like is a solution that works for *all possible* numbers of input values --- *which is an important reason to avoid using an array if you don't have to!!!*

Using expressions as array indexes

It is perfectly legal (and useful) to use expressions as indexes into an array.

For example we can assign values to *table[4], table[5],* and *table[6]* in the following ways.

```
int table[10];
int k;

k = 5;
table[k - 1] =  15;
table[k]     =  10;
table[k + 1]  =  5;
```

Swapping adajacent values in a array is also a useful thing to do in sorting operations (and in the codelab assignment);

```
int table[10];
int k;

k = 5;
if (table[k] > table[k + 1])
{
   table[k] = table[k + 1];
   table[k + 1] = table[k];
}
```

An incorrect approach to swapping

This approach *won't work!* By playing human computer we can see that the value of *table[k]*is destroyed and lost forever and both *table[k]* and *table[k + 1]* end up with the original value of *table[k + 1].*

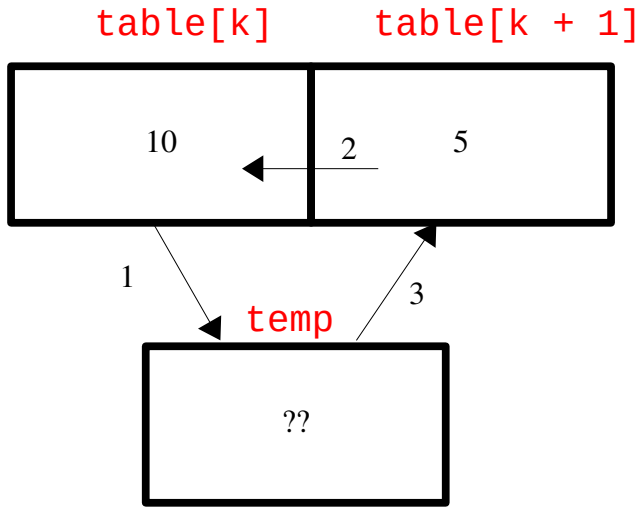
Exercise: If we interchange the order of the assignments will it fix the problem. Will the results be different but still broken?

A correct solution to the swapping problem

To avoid destroying the contents of one of the elements we *must use a temporary holding variable;*

```
int table[10];
int k;
int temp;

k = 5;
if (table[k] > table[k + 1])
{
[1]     temp = table[k];
[2]     table[k] = table[k + 1];
[3]     table[k + 1] = temp
}
```




```
[1]   temp gets the value 10
[2]   table[5] gets the value 5
[3]   table[6] getst the value 10
```

Designing a solution to the swapping problem in codelab

In this problem you must

- examine *each pair* of adjacent values in the array
- swap the adjacent values if they are out of order

When confronting problems such as this one it is very useful to

- start by solving a few examples by hand.

because

- Until you fully understand the problem and how to solve it manually you can't hope to be able to tell a computer how to do it!

- The examples you have solved by hand make useful test cases

From this example you might infer that all you need to do is move the first (or largest) value to the end of the array.

Initial state:
23 11 12 21 3 5
after k = 0
11 23 12 21 3 5
after k = 1
11 12 23 21 3 5
after k = 2
11 12 21 23 3 5
after k = 3
11 12 21 3 23 5
after k = 4
11 12 21 3 5 23

NOTE: There happen to be six elements in this array so *counter* = 6. But the largest value *k* can take on is 4.

But this one shows you that all the values can move.

4 19 3 2 25 1 7

Identifying and fixing errors

Regardless of your approach either your initial design or your initial implementation or both are likely to be WRONG.

The ability to discover what when wrong is just as important as the ability to design and implement the solution.

There are three rational approaches and all three can be useful:

- Play human computer and manually *walk through* your code noting all changes of state. The primary disadvantage of this approach is that *if you misunderstand the semantics of the language, your walk through will yield incorrect results.*

- Instrument your program with diagnostic prints directed to the standard error.

```
int table[10];
int k;
int temp;

if (table[k] > table[k + 1])
{
     fprintf(stderr, “swapping elements %d and %d \n”,
                        k, k + 1);
     temp = table[k];
     table[k] = table[k + 1];
     table[k + 1] = temp
     fprintf(stderr, “new value of table[%d] is %d \n”,
                        k, table[k]);
     fprintf(stderr, “new value of table[%d] is %d \n”,
                        k + 1, table[k + 1]);
}
```

  The main disadvantage of this technique is the possiblity for *data overload.* So you want to start with short test cases.

- Use *gdb (my favorite)*

Initializing scalars and arrays.

Both scalar values and arrays may be initialized when they are declared:

```
int counter = 0;
int sum = 0;
```

I recommend using this approach in *all programs you write.*

Beware of

```
int counter, sum = 0;
```

It will initialize *sum* but not *counter*!
Although C supports it, I recommend *against* declaring multiple variables in a single declaration.

When initializing an array,

- the initializers must be enclosed in {}
- you may provide fewer than the size of the array
- you may not provide *more* than the size of the array

```
int table[5] = {7, 9, 8};
```

Now

```
table[0] = 7, table[1] = 9, table[2] = 8, table[3] = ?, and
table[4] = ?
```

Using an array of counters

Suppose an input file contains a collection of single digit non-negative integers and I want to count how many 0's , 1's, 2's etc that there are in the collection. Your initial instinct may be to create a *maximally ugly* collection of *ifs*

```
if (val == 0)
    counters[0]= counters[0] + 1;
 else if (val == 1)
     counters[1] = counters[1] + 1;
 else if (val == 2)
```

The proper approach is surprisingly simple and should reside *forever* in your mental toolchest.

```
#include <stdio.h>

int main()
{
   int counters[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
                      // counters[0] counts 0's etc.
   int howmany = 0;  // howmany values were read
   int ndx = 0;      // array index

   howmany = fscanf(stdin, "%d", &ndx);

   while (howmany == 1)
   {
      counters[ndx] = counters[ndx] + 1;
      howmany = fscanf(stdin, "%d", &ndx);
   }
}
```

Processing character data

The char variable can hold the ASCII encoding of a single letter of the alphabet.

```
char x;
```

To read a character into a *char* variable the *%c* format code can be used. As with integers it is necessary to pass the address of the variable to *fscanf()*

```
howmany = fscanf(stdin, "%c", &x);
```

Here, each call to *fscanf()* consumes exactly one character. It will take 17 calls to consume the following data: 15 for the letters, 1 for the blank space (0x20) and one for the newline (0x0a).

```
Hello there
World
```

Printing the value of a character variable can be accomplished in an analogous way:

```
howmany = fprintf(stdout, "%c", x);
```

As with *ints* we pass values not addresses to *fprintf()*.

Single character variables have limited utility in a computer program. We might encode gender (M/F) in a single character but we often need to store whole words or phrases.

Words or phrases are commonly called *character strings* or simply *strings.*

Character arrays

- The C language has no character string data type.
- Instead character strings such as words and sentences are stored in character *arrays*.
- A byte having the value binary 0 indicates the end of the string.
- The %s format code is used to read/write strings with fscanf/fprintf

```
/* p13.c */
#include <stdio.h>

char word[6] = {'H', 'e', 'l', 'l', 'o', 0};

int main()
{
   fprintf(stdout, "%s\n", &word[0]);
}
```

```
class/101/examples ==> p13
Hello
class/101/examples ==>
```

Warning, leaving off the trailing 0 is fatal.

A useful shorthand:

The notation above is clearly tedious... so a handy substitute is provided.

Strings may be implicitly created using the *double quote operator* (as we have seen in format strings).

```
char word[] = “Hello”;
```

Reading and printing with the %s format.

As we saw earlier, individual whitespace characters are consumed one at a time while reading with the *%c* format code. The whitespace characters are *treated differently when using the %s format.* With the %s format each time a whitespace character appears *it is interpreted as a delimiter ending the current string*.

```
howmany = fscanf(stdin, "%s", word);
howmany = fprintf(stdout, "%s", word);
```

- *fscanf* will is read into the array location specifed. As before it returns the number of elements read. Here it will return 1 and *not the length of the string* if it is successful.
- *fscanf()* will automagically append the required byte of 0 used to indicate the end of memory resident strings
- *fscanf()* assumes that the the program has provided sufficient space to hold the entire string. If this is not true, *memory will be overwritten and the error may or may not be detected by the operating system.*

- *fprintf()* with the %s format code *does not print the 0 byte that terminates the string.*
- If the 0 byte is missing *fprintf()* will keep printing junk until it reaches a 0 byte!

- In contrast to %c and %d format codes the address of the string is passed to BOTH *fscanf() and fprintf()*

A string processing program

```
/* p16.c */

#include <stdio.h>

int main()
{
   char c[100];
   int  howmany;

   howmany = fscanf(stdin, "%s", &c[0]);
   while (howmany == 1)
   {
      fprintf(stdout, "%s \n", &c[0]);
      howmany = fscanf(stdin, "%s", &c[0]);
   }
   return(0);
}

p14.in
a b c d e

acad/cs101/examples/notes ==> p16 < p14.in
a
b
c
d
e
```

Multicharacter strings

We can see that this behavior persists even when multicharacter strings are used:

p16.in

```
       0 - 61 62 63 20 64 65 66 09 67 68 69 0d 6a 6b 6c 6d
            a  b  c     d  e  f     g  h  i     j  k  l  m
      10 - 6e 6f 0a 70 0a
            n  o     p
```

```
acad/cs101/examples/notes ==> p16 < p16.in
abc
def
ghi
jklmno
p
```

We can confirm that *fscanf()* will return the number of *complete strings read and not the number of characters in the string.* This will always be 1 when a format code containing a single %s succeeds. It is however possible to read more than one string in a single call to fscanf().

```
howmany = fscanf(stdin, "%s %s", word1, word2);
```

If this read succeeds a value of 2 will be returned.

Reading mixed string and integer data:

It is common to need to read a mixture of text and integer data. This is easily done *if and only if you know in advance where to expect text and where to expect integers*:

```
char id[16];
int dimension;

howmany =fscanf(stdin, "%s %d", &id[0], &dimension);
while (howmany == 2)
{
   process_input_value(&id[0], &dimension);
   howmany =fscanf(stdin, "%s %d", id, &dimension);
   fprintf(stdout, “%s %d\n”, id, dimension);
}
```

The input data:

```
width 400
depth 600
height 200
```

String Functions

- The C Standard Library contains a collection of functions used to manipulate strings.
- Some of the more useful ones are: *strcat, strchr, strcmp, strcpy, strcspn, strdup, strlen, strncat, strncmp, strncpy, strrchr, and strstr.*
- To use the C library functions be sure to include the function prototypes by doing:

  #include <string.h>

For details on how to use them use the *man* command.

```
man strchr

NAME
       strchr, strrchr, strchrnul - locate character in string

SYNOPSIS
       #include <string.h>

       char *strchr(const char *s, int c);

       char *strrchr(const char *s, int c);

RETURN VALUE
       The  strchr() and strrchr() functions return a pointer to the
       matched character or NULL if the character is not found.
```

Building a string function

To use the standard library function, just call it as you would a standard I/O function passing it proper parameters.

In lab we will construct our own version of some of them. The *strchr()* function searches a string for the occurrence of a particular character. If the character is found, its address is returned to the caller. Otherwise 0 is returned.

```
char *my_strchr(
char str[],
char c)
{
   int ndx = 0;

   while (str[ndx] != 0)
   {
      if (str[ndx] == c)
         return(&str[ndx]);
      ndx = ndx + 1;
   }
   return(0);
}
```

Command line arguments

It is often useful to pass arguments to a program via the command line. For example,

```
gcc -g -Wall -o p10 p10.c
```

- Elements of the command line are partitioned by whitespace characters.
- Each argument is always passed as a standard 0 terminated character string.

In this case the C compiler, *gcc,* is being passed 6 different arguments.

Argument	*Value*
0	gcc
1	-g
2	-Wall
3	-o
4	p10
5	p10.c

Printing command line arguments

When a program is started from the command line, the character strings (separated by whitespace) comprising the program name and the remaining arguments are copied by the Operating System into memory space occupied by the new program and a table or array of addresses is passed to the main function. These values can be accessed by the *main()* function as shown below.

```
/* printargs.c */

#include <stdio.h>

int main(
int argc,     /* number of command line arguments    */
char *argv[]) /* array of addresses of the arguments */
{
   int ndx = 0;

   while (ndx < argc)
   {
      fprintf(stdout, "%s\n", argv[ndx]);
      ndx = ndx + 1;
   }
}
```

When the program is invoked as follows:

```
==> printargs -Wall -o hello -g myprog.c
```

The following output is produced:

```
printargs
-Wall
-o
hello
-g
myprog.c
```

Processing numerical arguments

Suppose you mission is to write a program named *flag* whose function is to produce a .ppm image of a flag. Your program is to be invoked as:

```
flag width-of-flags-in-pixels
---
flag 800
```

- The value 800 is passed to your program as a character string consisting of three bytes of data: 0x38, 0x30, 0x30
- To use the value in your program you must convert it to an integer.
- The *sscanf()* function scans from a *string* instead of a file and can do what you need.

```
#include <stdio.h>
#include <stdlib.h>

int main(
int argc,    /* number of cmd line args */
int *argv[]) /* array of arg addresses  */
{
   int width = 0;
   int howmany = 0;

   if (argc < 2)
   {
      fprintf(stderr, "usage is flag width-in-pix\n");
      exit(1);
   }

   howmany = sscanf(argv[1], "%d", &width);

   fprintf(stderr, "Width = %d \n", width);
}
```

Structured data types

So far we have considered only basic data types (*int, char)* and arrays of basic data types. It is often useful to have a mechanism by which a programmer may create new data types which are aggregations of previously existing types. The *structure* provides us this capability in the C language.

A C structure is declared as follows:

```
struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};
```

- Note that *struct pixel_type is the name of a programmer defined structure data type.*
- It is *not* the name of a variable.
- The use of *_type* as a suffix is not required but can help you remember what name is a type and what name is a variable.

The variables comprising the structure (r, g, b) are known as members or elements.

Creating an instance of a structured.

To create an instance of an *int* or *char* type we use the type name followed by a programmer selected variable name:

```
int sum;
char code;
```

The same approach is used with structure types. You just give the type name and follow it by a variable name of your choosing.

To declare an *instance* (*variable*) of type *struct pixel_type* use:

```
struct pixel_type pixel;
```

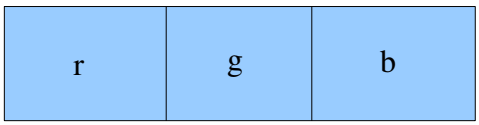
`struct pixel_type`	is the name of the *type*
`pixel`	is the name of a *variable* of type *struct pixel_type*

To set or reference components of the structure *pixel* use the form:

```
structure-instance-name.element-name

pixel.r = 250;  // make Mr. Pixel yellow
pixel.g = 250;
pixel.b = 0;
```

The structure variable *pixel* occupies 3 bytes of storage.



A bad (but syntactically legal) idea.

The C language is sensitive to the context in which a name is used and so the following will not cause compile errors, *but its a very bad practice.*

```
struct pix
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct pix pix;
```

Using the *typedef* facility

Because it can be painful to have to type the word *struct* over and over and over... The C language makes it possible to create a structured type and give it a *standalone* type name.

```
typedef struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
} pixel_t;
```

A type name created with *typedef* may be used just like a primitive type in creating instances of the structured type.

```
pixel_t newpix;
```

Arrays of structures

We can also create an array of stucture types .

```
struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct pixel_type pixmap[600 * 800];
```

To access an individual element of the array place the subscript next to the name it indexes

```
pixmap[15].r = 250;
```

Structures and Arrays within structures

It is common for structures to contain elements which are themselves structures or arrays of structures. In these cases *the structure definitions should appear in "inside-out" order.* This is done to comply with the usual rule of not referencing a name before it is defined. Since the *image_type* structure contains an element which is a *pixel_type* structure. The *pixel_type* definition *must* appear first in the source code.

```
typedef struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
} pixel_t;
```

For the major project in the course will be be constructing an image manipulation program. We will be using a structure of this form:

```
typedef struct image_type
{
   char filename[64];     /* name of disk resident file */
   char id[4];            /* P5 gray or P6 color        */
   int  width;            /* Width in pixel cols        */
   int  height;           /* depth in pixel rows        */
   pixel_t pixels[1024*1024]; /*  The pixel data        */
}  image_t;
```

As before we can create an instance of our image structure by declaring:

```
struct image_type input;
```

or

```
image_t output;
```

Each one of these variables occupies $64 + 4 + 4 + 4 + 3 * 2^{20} = 3{,}145{,}804$ bytes of storage!!!

It should be clear that we cannot process images having more than 1024 * 1024 pixels using this structure!

Accessing elements of the image_t structure

Elements of the structure maybe accessed as described before:

```
output.width  = 640;
output.height = 480;
output.pixels[0].r = 255; // make first pixel yellow
output.pixels[0].g = 255;
output.pixels[0].b = 0;
```

For determining where to put the [] and the . use the following rule:

Subscripts are required in code

- immediately following any name declared as an array
- and are allowed no where else!

You might also be tempted to try something like:

```
output.filename  = "image1.ppm";
```

However, the C language has no string assignment capability and *you will get a compiler error*.

You can read filename from the standard input using:

```
howmany = fscanf(stdin, "%s",  &output.filename[0]);
```

or copy a string to it by using:

```
strcpy(&output.filename[0],  “image1.ppm”);
```

Addresses of structures:

Because structures can be so very large

- It is inefficient to pass them as parameters from one function to another.
- Therefore we typically pass the *address of the structure* instead of the structure itself.
- Passing the address of a structure also make it possible for the called function to *modify elements the stucture.*

We commonly call a variable that holds the address of another variable a *pointer*

To declare an *address / pointer variable*

- we begin the declaration with the *name of the type pointed* to
- but to tell the compiler this is a pointer to the structure and not an instance of the structure we *preface the variable name with the* * character

```
struct pixel_type *pixptr;
```

or (assuming we used *typedef*)

```
pixel_t *pixptr;
```

To access elements of the structure via a pointer we use the -> operator instead of the . operator.

```
pixel_t *redptr;
pixel_t  redpix;

redptr->r = 255;
redptr->g = 0;
redptr->b = 0;
```

but

```
redpix.r = 255;
redpix.g = 0;
redpix.b = 0;
```

Example of a complete program

The structure definition *must* precede any use of the structure type!

```
#include <stdio.h>
typedef struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
} pixel_t;

void make_cyan_pixel(pixel_t *pixptr);

int main()
{
   pixel_t pixel;   // create an instance of a pixel
   make_cyan_pixel(&pixel); // pass address to function
   printf("%d %d %d \n", pixel.r, pixel.g, pixel.b);
}
```

Since the *make_cyan_pixel()* expects a pointer to the structure we must pass the address of the structure here

Accessing structure elements via a structure pointer

To access an element of a structure using a pointer we use - > instead of .

Since the *main()* function is dealing with an *instance* and *make_cyan_pixel()* is dealing with a pointer they must access the elements differently.

```
void make_cyan_pixel(
pixel_t *pixptr)
{
    pixptr->r = 0;      // make Mr. pixptr-> cyan
    pixptr->g = 250;
    pixptr->b = 250;
}
```

Using a 1-dimensional array to represent 2 - dimensional data

A *.ppm* image is logically a 2-dimensional entity. The origin is at the upper left corner. The positive *x-axis* is horizonal pointing to the right. The positive *y-axis* is vertical *pointing downward.*

Thus the (x, y) coordinates of a pixel are often also expressed as (row, col) coordinates where col = x and row = y. Note that (x, y) and (row, col) notation are by convention inconsistent :-(

Suppose the integer variables *width* and *height* represent the number of columns and rows in the image. Suppose a grayscale image is being constructed in the following array:

```
unsigned char image[300 * 200];
int width = 200;
int height = 300;
int ndx;
```

If we wish to compute the index of the pixel at location *(row, col)* within the image the value of *ndx* should be computed as:

```
ndx = row * width + col;
```

thus we can refer to the grayscale level of the pixel at location (row, col) either as:

```
image[ndx]
```

or

```
image[row * width + col];
```

For example, if the value of width is 10, then there are 10 pixels per image row. To reach the pixel whose (row, column) address is (3, 5) it is necessary to pass over three complete rows (row 0, row 1, and row 2) and 5 pixels in row three (pixels 0, 1, 2, 3, and 4).

Thus, the offset of the pixel at (3, 5) is 3 * 10 + 5 as shown above.

Mapping the offset within an image to (*row, col)*

If we know the value of *ndx*, the offset within an image but wish to compute the value of *row* and *col*, we can divide both sides of the equation by *width* and see that

```
          row remainder col
width | ndx
```

In integer arithemetic in C this becomes

```
row = ndx / width;
col = ndx – row * numcols = ndx % width;
```

In C, the operator % performs the *mod* function.

`a % b` is the remainder when a is divided by b.

For example `17 % 5 is 2.`

The result of `a % b` is an integer in the range `0, 1, 2, ... b-1`

Thus one way to build a grayscale picture is:

```
ndx = 0;
while (ndx < width * height)
{
    row = ndx / numcols;
    col = ndx % numcols;
    image[ndx] = pix_value(height, width, row, col);
    ndx += 1;
}
```

Here *pix_value( )* is a user written function. It is passed

- height, width and row, col coordinates of the pixel and
- computes and returns a gray scale value

Revisiting building a .ppm file

In this example we will revisit the example on page 115 of the notes using

- our structure definitions and
- block output

```
/* p12c.c */

/* Construct a continuously varying color image          */
/* The red component is a function of pixel row          */
/* The blue component is a function of pixel col         */
/* The blue component is 0                               */
/* Input data                                            */
/*     width  in pixels  height in pixels                */
/*     red value  green value  blue value                */
```

```
#include "image.h"
```

Programmer written .h file containing system includes, structure definitions and function prototype definitions

The image.h file

It will be convenient to group structure definitions and function prototypes into a single source file that can be included by multiple .c files.

```
/* image.h */

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>

typedef struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
} pixel_t;

typedef struct image_type
{
   char filename[64];     /* name of disk resident file */
   char id[4];            /* P5 gray or P6 color        */
   int  width;            /* Width in pixel cols        */
   int  height;           /* depth in pixel rows        */
   pixel_t pixels[1024*1024]; /* The pixel data         */
}  image_t;


void make_ppm_row(image_t *image, int row);
void make_ppm_image(image_t *image);
void make_ppm_pixel(image_t *image, int row, int col);

void write_ppm_image(image_t *image);
void write_ppm_header(FILE *out, image_t *image);
```

Including system include files within the programmer provided include file allows us to avoid having to manually add them to each source code module.

The main() function

As before the main function reads the dimensions from the standard input, but now it must store them in the image structure.

```
int main(
int argc,
char **argv)
{
   image_t output;

/* Make sure command line parameters were provided */

   if (argc < 3)
   {
      fprintf(stderr, "Usage is a.out width height \n");
      exit(1);
   }

/* Read image dimensions from command line */
/* In the real world we should verify these are sensible */

   sscanf(argv[1], "%d", &output.width);
   sscanf(argv[2], "%d", &output.height);

   strcpy(&output.id[0], "P6");
   strcpy(&output.filename[0], "out1.ppm");

/* Create the image file */

   make_ppm_image(&output);

/* Write the image data to the output file */

   write_ppm_image(&output);

   return(0);
}
```

Constructing the ppm image

A one-row-at-a-time approach can avoid nested looping

```
void make_ppm_image(image_t *img)
{
   int row = 0;

/* build the image  one row at a time */

   for (row = 0; row < img->height; row += 1)
   {
      make_ppm_row(img, row);
   }
}
```

Constructing a single row of the output image

```
void make_ppm_row(image_t *img, int row)
{
   int col = 0;

   for (col = 0; col < img->width; col += 1)
   {
      make_ppm_pixel(img, row, col);
   }
}
```

Constructing a single pixel of the output image

Instead of using *fprintf()* to directly print the pixels we construct the entire pixel array in the image structure in memory! The variable *ndx* is used to access the correct pixel structure. Since the row and column value *always start with 0* the correct offset in the one-dimensional pixel array is

ndx = *row* * *width* + *column*

```
void make_ppm_pixel(image_t *img, int row, int col)
{
   int ndx = row * img->width + col;

   img->pixels[ndx].r = 255 * row / img->height;
   img->pixels[ndx].g = 255 * col / img->width;
   img->pixels[ndx].b = 0;
}
```

A more complex image

```
void make_ppm_pixel(image_t *img, int row, int col)
{
   int ndx = row * img->width + col;
   float radius = 0.3 * img->height;
   float cx, cy;
   float dx, dy;
   float dist;

/* Compute the coordinates of the center pixel */

   cy = img->height / 2;
   cx = img->width  / 2;

/* Compute x and y distances of this pixel from center */

   dx = col – cx;
   dy = row – cy;

/* Compute distance of this pixel from the center */

   dist = sqrt(dx * dx + dy * dy);

   img->pixels[ndx].r = 255;
   img->pixels[ndx].g = 255;
   img->pixels[ndx].b = 255;

/* If pixel is inside circle of radius 0.3 * height */
/* make it red.                                     */

   if (dist <= radius)
   {
       img->pixels[ndx].g = 0;
       img->pixels[ndx].b = 0;
   }
}
```

Exercise: What does the above image look like??

Writing the ppm image

Instead of writing to the standard output, this program writes directly to a disk file of a specfied name. To do this the *fopen()* function must be used to open the file. The *fwrite()* function is used to write the entire image in a single operation. This is much more efficient that using *fprintf()* for every pixel.

```
void write_ppm_image(image_t *img)
{
   int row = 0;
   int howmany = 0;
   FILE *out;

/* Open the target output file for writing and */
/* verify success                              */

   out = fopen(img->filename, "w");
   if (out == 0)
   {
      fprintf(stderr, "Couldn't open %s \n",
                              img->filename);
      exit(1);
   }

/* Write the ppm header */

   write_ppm_header(out, img);

/* Write the whole image in a single function call */

   howmany = fwrite(img->pixels, sizeof(pixel_t),
                 img->height * img->width, out);
   if (howmany != img->height * img->width)
   {
      fprintf(stderr, "Write error %d \n",
                              howmany);
      exit(1);
   }
}
```

Parameters of *fopen()*

The first parameter that is passed to *fopen()* must 0 terminated character string that specifies the name of a file on disk.

If the name does not start with / , it is assumed to be relative to the current working directory. Suppose we wish to read or write a file named *in.txt* that lives in directory */home/westall/projects*

Possible ways of specifying the file name include

```
"/home/westall/projects/in.txt"
```
-- works regardless of current working directory

```
"projects/in.txt"
```
-- works if and only if the current working directory is */home/westall*

```
"in.txt"
```
-- works if and only if the current working directory is /home/westall/projects

The second parameter is the *access mode.* Legal values include

`"r"` – read only
"w" – write only
"rw" – read and write
"a" – append to end of existing file

In the M$ world a b (for binary) may be appended to *disable* mangling of \r and \n data – "rb"

*The fopen() function returns a FILE * address variable (pointer) which must be assigned to the local variable you will use to access the file.*

The type FILE is created using a *typedef* in *stdio.h* but the programmer doesn't need to know the internal details of the structure. This type of structure is sometimes called an *abstract data type (ADT).*

Block input and output

The *fread()* and *fwrite()* functions are the most efficient way to read or write large amounts of data.

```
howmany = fread(location,  element_size,
                number_of_elements,  file_ptr);
howmany = fwrite(location,  element_size,
                number_of_elements,  file_ptr);

howmany = fwrite(img->pixels, sizeof(pixel_t),
            img->height * img->width, out);
```

- The first parameter is the address of the data to be written or memory area to be read into.
- The second parameter passed to the function is the *size of a basic data element* to be read or written
- The third parameter is the *number of elements*. to be read or written.
- The forth parameter is the pointer to the FILE structure that was returned by *fopen()*.

Writing the .ppm header

The only changes needed are to

be sure to write to the file *out* and not *stdout*

use the elements if the image_t structure

```
void write_ppm_header(FILE *out, image_t *im)
{
   fprintf(out, "%s\n", im->id);
   fprintf(out, "%d %d 255\n", im->width, im->height);
   return;
}
```

Using dynamically allocated pixel array storage

A disadvantage of the approach used in the previous program is that images larger than 1 megapixel cannot be created without changing the *image_t* structure and recompiling the program. We can fix this by using a dynamically allocated array of pixels.

We replace the pixel array in the image structure with an address variable that points to an array of pixels as shown below:

```
typedef struct image_type
{
   char filename[64];      /* name of disk resident file */
   char id[4];             /* P5 gray or P6 color        */
   int  width;             /* Width in pixel cols        */
   int  height;            /* depth in pixel rows        */
   pixel_t *pixels;        /* Address of the pixel data  */
}  image_t;
```

- This structure now occupies only 80 bytes!!
  and
- We can dynamically create the pixel array to have exactly the correct size.

Dynamic allocation of the pixel buffer

The only code change needed is to dynamically allocate the pixel array using the *malloc()* function which allocates the requested array and returns its address. The remainder of the program is completely unchanged!

```
int main()
{
   image_t output;

/* Read image dimensions and pixel color */

   fscanf(stdin, "%d %d",
         &output.width, &output.height);

   output.pixels = (pixel_t *)malloc(sizeof(pixel_t) *
                        output.width * output.height);

   if (output.pixels == 0)
   {
      fprintf(stderr, "Malloc failed \n");
      exit(1);
   }
}
```

Use of *sizeof()* with structures

The *sizeof()* facility should *always* be used in dynamically allocation storage for structured data types and in reading and writing them.

If the size of the structure changes as the program evolves, then calls to *malloc(), fread(), fwrite()* will (generally) adapt correctly.

However, it is somewhat easy to do this incorrectly.

```
struct pixel_type
{
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct pixel_type pix;
struct pixel_type *pixptr;
```

The following all have the value 3

```
sizeof(pix)
sizeof(struct pixel_type)
sizeof(*pixptr)
```

The following has the value 4

```
sizeof(struct pixel_type *)
sizeof(pixptr)
```

Confusing these is a common source of some *very nasty* program bugs.

Reading ppm image

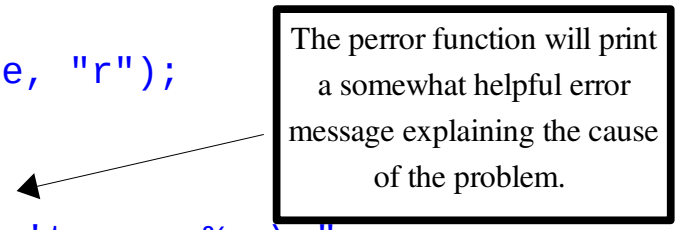
When reading a *.ppm* file it is necessary to:

- verify that the file contains a color (P6) ppm image
- save the dimensions in the image_t structure
- allocate the pixel buffer
- read the pixel data into the pixel buffer

```
void read_ppm_image(
image_t *image)
{
   FILE *in;
   int  howmany = 0;

/* Open the target output file for reading and */
/* verify success                              */

   in = fopen(image->filename, "r");
   if (in == 0)
   {
      perror("open failed");
      fprintf(stderr, "Couldn't open %s \n",
                      image->filename);
      exit(1);
   }
```



```
/* Read the ppm header */

   read_ppm_header(in, image);

/* Allocate the pixel buffer */

   image->pixels = (pixel_t *)malloc(sizeof(pixel_t) *
                        image->height * image->width);

/* Write the data and ensure correct amount written */

   howmany = fread(image->pixels, sizeof(pixel_t),
                image->height * image->width, in);

   if (howmany !=  image->height * image->width)
   {
      fprintf(stderr, "Read error wanted %d got %d \n",
                     image->height * image->width,
                     howmany);
      exit(1);
   }
}
```

Reading the ppm header

- The header is not fixed format.
- It can contain any number of comment lines.
- All comment lines must begin with #

```
void read_ppm_header(
FILE *in,
struct image_type *image)
{
   int  vals[3];   // temporarily holds width, height, 255
   int  count = 0; // numeric values read so far
   char buf[256];  // trash can
   int  howmany;
   int  not_toomany = 1000;  // loop limiter

/* Read the P6 (or P5) if grayscale */

   howmany = fscanf(in, "%s",  &image->id[0]);
   if ((howmany != 1) || (image->id[0] != 'P'))
   {
      fprintf(stderr, "read_ppmhdr bad magic number \n");
      exit(1);
   }
```

```
/* Now consume the width, height, and maximum pixel  */
/* value from the header... fscanf() may fail due   */
/* to comments in the header.. When it does fgets() */
/* will consume the remaining data on the line.     */

   while ((count < 3) && (not_toomany))
   {
      howmany = fscanf(in, "%d", &vals[count]);
      if (howmany == 0)
         fgets(buf, 256, in);
      else
         count = count + 1;
      not_toomany = not_toomany - 1;
   }

/* If something bad went wrong, abort now */

   if ((not_toomany == 0) || (count != 3))
   {
      fprintf(stderr, "read_ppm_header: broken header \n");
      exit(1);
   };

/* This consumes the byte of whitespace following 255  */
/* and saves the image dimensions in the proper places */

   fscanf(in, "%c", &buf[0]);

   image->width = vals[0];
   image->height = vals[1];

}
```

Exercise: *Create a program that will make a mirror image of the input image. It should consist of 4 separate files: image.h (described previously); image.c (contains read_ppm_image(); and write_ppm_image()); main.c (given below) and mirror.c (described below).*

```
#include "image.h"

int main(
int argc,
char *argv[])
{
   image_t input;
   image_t output;

   strcpy(&input.filename[0], argv[1]);
   strcpy(&output.filename[0], argv[2]);

   read_ppm_image(&input);

   mirror_ppm_image(&input, &output);

   write_ppm_image(&output);

}
```

The *mirror.c* file

For all problems it is useful to view the process as building the *output image* one pixel at a time. Start at the top of the image (row 0) and proceed across and down. The value of row and column will always represent the current location in the *output image*. The basic structure here follows the previous example: *make_ppm_image()*.

```
void mirror_ppm_image(
image_t *inimg,
image_t *outimg)
{
```

Before processing rows of the output image, it is necessary to

- copy width, height and id from inimg to outimg
- allocate the pixel buffer for the output image

```
}

void mirror_ppm_row(
image_t *inimg,
image_t *outimg,
int row)    // current row in input and output image
{

}

void mirror_ppm_pixel(
image_t *inimg,
image_t *outimg,
int row,    // current row in input and output image
int col)    // current column in OUTPUT image
{


}
```

Structures as function parameters and return values

Passing structures, especially pointers to structures, is a powerful way for one function to pass a *large amount of information* to another *using a small number of parameters*.

Structures as parameters

The example program below illustrates two aspects of passing structures

- A *struct*, like an *int* may be passed as a parameter to a function.

In fact the process works just like passing an *int* in that:

- The complete structure is *copied* onto the stack – which is *very inefficient* for a large structure.
- The function is *unable* to modify the caller's copy of the structure variable

The *try_to_mod()* function – structure version

This example demonstrates that a called function cannot modify the caller's copy.

```
struct work_type
{
   int w;
};

void try_to_mod(struct work_type q)
{
   printf("q.w = %d \n", q.w);
   q.w = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   try_to_mod(z);
   printf("z.w = %d\n", z.w);
}

acad/cs101/examples/notes ==> gcc -o p41 p41.c
acad/cs101/examples/notes ==> p41
q.w = 99
z.w = 99    <--- value was not modified

acad/cs101/examples/notes ==> cat p41.c
```

Avoid passing large structures as parameters!!

Another disadvantage of passing structures by value is that copying large structures onto the stack

- is very inefficient and
- may even cause program failure due to stack overflow.

```
   struct work_type
   {
      int w[1024 * 1024];
   };

/* This fellow will cause a total of 4 Terabytes  */
/* to be copied onto the stack.                   */

   struct work_type fourMB;

   for (i = 0; i < 1000000; i++)
      slow_call(fourMB);
```

Note that in the C language it is *not possible to pass an array by value on the stack.*. Thus the only way to produce this behavior is to embed the array in a structure!

Passing pointers to structures

In well-written C programs it is much more common to pass a pointer to a structure rather than passing the entire structure. Two principle advantages of doing so include:

- Passing a pointer requires that only a single word by pushed on the stack *regardless of how large the structure is.*
- The called function can now modify elements of the structure.
- It is possible for some elements of the structure to be input parameters (e.g. *img->filename) while*
- Other elements may be output values (*img->width, img->depth)* to be filled in by the called function.
- When a pointer is passed, the called function must use -> instead of . to access structure elements.

The *try_to_mod()* function with pointer to structure

As was the case when an array address is passed to a function, the structure can also be modified when the address of the structure instead of the structure itself is passed.

```
/* p42.c */

struct work_type
{
   int w;
};

void fun(struct work_type *q)
{
   printf("q->w = %d \n", q->w);
   q->w = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   fun(&z);
   printf("z.w = %d\n", z.w);
}
acad/cs101/examples/notes ==> gcc -o p42 p42.c
acad/cs101/examples/notes ==> p42
q->w = 99
z.w = 1000   <---- value has been modified
```

The *const* qualifier

But what if you *don't want* the recepient to be able to modify the structure? Then the *const* qualifier can be prefixed to the parameter.

```
/* p43.c */

struct work_type
{
   int w;
};

void fun(const struct work_type *q)
{
   printf("q->w = %d \n", q->w);
   q->w = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   fun(&z);
   printf("z.w = %d\n", z.w);
}
```

```
acad/cs101/examples/notes ==> gcc -g p43.c
p43.c: In function `fun':
p43.c:11: error: assignment of read-only member `w'
```

This form of protection is not especially strong and is easily bypassed

```
/* p44.c */

struct work_type
{
   int w;
};

void fun(const struct work_type *q)
{
   struct work_type *local;

   local = (struct work_type *)q;

   printf("q->w = %d \n", q->w);
   local->w = 1000;
}

int main()
{
   struct work_type z;

   z.w = 99;
   fun(&z);
   printf("z.w = %d\n", z.w);
}

acad/cs101/examples/notes ==> gcc -g p44.c -o p44
acad/cs101/examples/notes ==> p44
q->w = 99
z.w = 1000
```

Structures as return values from functions

- Scalar values (*ints, floats, etc)* are efficiently returned in CPU registers
- Historically, structure assignments and the return of structures were not supported in C.
- But the return of *pointers* including *pointers* to structures has always been supported.
- Now both structure assignment and structure returns are supported

```
/* p31.c */

struct work_type
{
   int w;
};

struct work_type *fun(void)
{
   struct work_type work;

   work.w = 99;
   return(&work);
}

int main()
{
   struct work_type *q;

   q = fun();
   printf("%d \n", q->w);
}

acad/cs101/examples/notes ==> gcc -o p31 p31.c
p31.c: In function `fun':
p31.c:13: warning: function returns address of local
variable
acad/cs101/examples/notes ==> p31
99
```

The reason for the warning is that the function is returning a pointer to a variable that was allocated on the stack during execution of the function. Such variables are subject to being wiped out by subsequent function calls.

This example shows this can indeed occur.

```
/* p32.c */

struct work_type
{
   int w;
};
struct work_type *fun(void)
{
   struct work_type work;

   work.w = 99;
   return(&work);
}

int fun2(int v)
{
  int w;
  int x;
  w = v;
  x = w * v;
  return(x);
}

int main()
{
   struct work_type *q;
   int z = 12;
   q = fun();
   z = fun2(z);
   printf("%d \n", q->w);
}
```

```
acad/cs101/examples/notes ==> gcc -o p32 p32.c
p32.c: In function `fun':
p32.c:13: warning: function returns address of local variable
acad/cs101/examples/notes ==> p32
-1077017368
```

Return of structures

It is possible for a function to return a structure. This facility depends upon the structure assignment mechanisms which copies one complete structure to another.

- This avoids the unsafe condition associated with returning a pointer but
- incurs the possibly extreme penalty of copying a very large structure

```
/* p33.c */

struct work_type
{
   int w;
};

struct work_type fun(void)
{
   struct work_type work;

   work.w = 99;
   return(work);
}

int main()
{
   struct work_type q;
   struct work_type z;
   q = fun();
   z = q;
   printf("%d %d \n", q.w, z.w);
}
```

```
acad/cs101/examples/notes ==> gcc -g -o p33 p33.c
acad/cs101/examples/notes ==> p33
99 99
acad/cs101/examples/notes ==>
```

Summary

Passing/returning instances of structures potentially incurs big overhead
Passing/returning pointers incurs almost no overhead
Accidental modifications can be prevented with *const*

- Therefore, it is recommended that you *never pass nor return* an instance of a structure unless you have a very good reason for doing so.

This problem doesn't arise with arrays.

- The only way to pass an array by value in the C language is to embed it in a structure!!
- The only way to return an array is to embed it in a structure

Floating point

Unlike int, floating point values can represent fractional entities and are thus more useful in most scientfic computations.

There are two types of floating point variables in the C language.

```
float        32 bits
double       64 bits
```

Example declarations

```
float a;
double c;
```

Floating point constants can be expressed in two ways

Decimal number	`1024.123`
Scientific notation	`1.024123e+3`

```
avogadro = 6.02214199e+23;
```

Format codes for floating point input and output:

There are three basic specifiers e, f, and g. Each *must* be prefaced by the letter *l* if a double precision value is being read or printed. The usage of each of the 6 possible codes in output is shown below.

%f	single precision floating point number *(float)* in decimal	notation
%lf	double precision floating point number *(double)* in decimal	notation
%e	single precision in scientific notation	
%le	double precision in scientific notation	
%g	single precision with auto notation selection	
%lg	double precision with auto notation selection	

The size of an output field and the number of digits to the right of the decimal may also be specified in output format strings. Specifying both field width and decimal digits is useful and often necessary when trying to make output values "line up" properly in columns. The *modifiers should not be used for input.*

Floating point input

For input it is not necessary to distinguish, e, f, and g. Any of the codes can be used to read decimal or scientific input values. It is necessary to use *l* if reading double precesion.

```
int howmany;
double fpvalues[2];

howmany = fscanf(stdin, "%lf %lf", &fpvalue[0],
                          &fpvalue[1]);
```

Floating point output

```
howmany = fprintf(stdout, "%8.3lf %8.3lf \n",
                  fpvalue[0], fpvalue[1]);
```

%8.3lf means print with field width of 8 characters with 3 digits to the right of the decimal

Internal representation of floating point numbers

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them.

Single Precision

The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right.

- The first bit is the sign bit, S, 1 means negative and 0 means positive.
- the next eight bits are the exponent bits, 'E',
- and the final 23 bits are the fraction 'F':

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
0 1      8 9                     31
```

The value V represented by the word may be determined as follows:

- If E=255 and F is nonzero, then V=NaN ("Not a number")
- If E=255 and F is zero and S is 1, then V=-Infinity
- If E=255 and F is zero and S is 0, then V=Infinity
- If 0<E<255 then V=(-1)**S * 2 ** (E-127) * (1.F) where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then V=(-1)**S * 2 ** (-126) * (0.F) These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

Specific examples

```
0 00000000 00000000000000000000000 = 0
1 00000000 00000000000000000000000 = -0

0 11111111 00000000000000000000000 = Infinity
1 11111111 00000000000000000000000 = -Infinity

0 11111111 00000100000000000000000 = NaN
1 11111111 00100010001001010101010 = NaN

0 10000000 00000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2
0 10000001 10100000000000000000000 = +1 * 2**(129-127) * 1.101 = 6.5
1 10000001 10100000000000000000000 = -1 * 2**(129-127) * 1.101 = -6.5

0 00000001 00000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-126)
0 00000000 10000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-127)
0 00000000 00000000000000000000001 = +1 * 2**(-126) *
                                     0.00000000000000000000001 =
                                     2**(-149)  (Smallest positive value)
```

Floating point versus integer

- Integer values are equally spaced on the number line.
- The distance between any two adjacent ints is 1

This is not true of floating point

- One half of *all* the diffrent floating point numbers lie between 0 and 1!
- The largest positive int is 2^31 – 1.
- The largest postitive float is on the order of 2^127
- Floats even smaller than the largest positive *int* have greater than integral spacing!

```
#include <stdio.h>

int main()
{
   float x;
   float y;

   x = 123456789;
   y = 123456791;

   printf("%12.0f %12.0f \n", x, y);
}
 ==> ./a.out
   123456792    123456792
```

This problem occurs because while positive ints have 31 significant bits, *floats* have only 32-9 = 23.

Double precision

The problem can be rectified by the use of *double precision* variables.

They are declared as follows:

```
double x;
```

The are read or printed using the *%lf, %le, or %lg f*ormat codes.

It is *very important* the %lf family of format codes be used with and *only with* doubles.

Mixing *float* and *int* in an expression

- Different sets of hardware components are used to perform integer operations and floating point operations.

- NO HARDWARE exists that can *add, subtract, multiply. or divide* an integer operand and a floating point operand.

- Therefore, whenever an expression contains mixed mode (integer and floating point operands) the *integer operand is always invisibly converted to floating point* before the operation is performed.

- A mixed mode example.

```
int x;

x = (5 / 2.0) * 4;
```

- If an integer 2 were used in this expression, its value would be 8.

- Since 2.0 is a floating point number

    - The 5 will be converted to floating point and the result of the division will be 2.5
    - Then the 4 will be converted to floating point and the final result will be 10.0
    - Finally the 10.0 will be converted back to *int 10* before being assigned with *x.*

Parsing

Parsing is the process of consuming elements of some input language and mapping the information contained therein to a usable data structure. We will define an input language for specifying operations on images and build a parser that can digest the language.

An example of a "sentence" in our language is:

```
fade
{
   in0  dive.ppm
   factor 0.4
   out faded_dive.ppm
}
```

This alternative variant specifies the same information.

```
fade
{
   factor      0.4
   in0         dive.ppm
   out         faded_dive.ppm
}
```

The following rules define our language

- Words in a sentence must be separated by one or more whitespace characters.
- Each sentence starts with the name of an image operation (fade, gray, mirror, blend, etc...)
- The operation must be followed by the { character.
- Following the { are a variable number of clauses of the form parameter-name parameter-values
- The parameter names (like image operations) are pre-defined reserved words in the language.
- For each parameter name, the type and number of parameter values is always the same.
- A parameter-name parameter-values clause may be followed by either another parameter-name parameter-values clause or the } character which indicates the end of a sentence.

Implications of the rule set

- The fact that words in the language *must* be separated by white space makes it possible to use the *fscanf(stdin,...)* function with the %s, %d, %lf format codes to consume text, integers and double precision values respectively.
- We cannot assume anything about the order in which the parameter-name parameter value sets appear.
- We have a potential need to deal with missing parameters (e.g. *factor* missing in a *fade* operation) or
- Extraneous parameters (e.g. *factor* specified) in a *gray* operation.

The target data structure

For consistency and ease of remembering we use the *same names* for each parameter in both our language and its corresponding element in the following structure.

```
typedef struct imageop_type
{
   int    opcode;    // index of operation name
   char   in0[64];   // primary input image
   char   in1[64];   // secondary input image
   char   out[64];   // output image
   int    dims[2];   // dims of output for resize
   double factor;    // factor for gray or blend
}  imageop_t;
```

Table lookups

Table lookups will comprise an important element of the processing.

A table of addresses of character strings can be constructed as follows:

```
char *first_names[] =
{
   "michael",
   "jeffrey",
   "shannon",
   "logan",
   "joshua",
};

char *op_names[] =
{
   "mirror",
   "gray",
   "fade",
};

char *param_names[] =
{
   "in0",
   "in1",
   "out",
   "dims",
   "factor",
};
```

The *table_lookup()* function

You will build a table lookup function whose mission is to return the index of the parameter string name in the parameter table. It should return (-1) if the string is not in the table.

The *strcmp()* function from the standard string library should be used to compare the string pointed to by *table[i]* with the string *name* for i = 0 to count - 1.

```
int table_lookup(char *table[], int count, char name[])
{

}
```

Invoking table lookup:

```
int count = sizeof(first_names)/sizeof(first_names[0]);

int ndx = table_lookup(first_names, count, "shannon");

int ndx = table_lookup(first_names, count, "gray");
```

- In the first case the value of *ndx* should be set to 2.
- In the second case it should be set to -1.
- Setting count as shown is important.. It makes it easy to add or remove table entries without having to change the code that invokes table_lookup.

Building the complete parser

An invocation of the parser will attempt to read *exactly one sentence* in our input language. The parser will read from the standard input (*stdin).* Later we will want to be able to perform multiple image operations in a single run of our program, but that will require multiple invocations of the parser!

```
int parser(imageop_t *op)
{
  read a string from stdin into into  character array opname;
  if (no data is read)    // end of file reached
      return(-1);
  attempt to look up opame in the op_names[] table
  if look up fails print the invalid opname and exit.
  save the table index that was returned in op->opcode.
  invoke consume_parameters(op) to consume the delimiters { }, parameter names,  and
  parameter values.
  return(0);
}
```

Consuming the parameters of a particular operation

```
int consume_parameters(imageop_t *op)
{
  read a string from stdin into into  character array pname;
  ensure that pname[0] is the { character.
  If not print pname in an error message and exit

  read a string from stdin into into  character array pname;
  while (pname[0] is not the } character and not end-of-file)
  {
    attempt to look up pname in the param_names[] table
    if that fails print the invalid pname in an error message and exit.
     consume_parameter_values(op, table_index);
     read next string from stdin into into  character array pname;

  }
  return(0);
}
```

Consuming the values associated with a parameter

There exist far more elegant ways to consume the parameter values, but the simplest way is via a big switch statement:

```
int consume_parameter_values(imageop_t *op, int ndx)
{
   int howmany;

   switch (ndx)
   {
   case 0:   // param is in0
      howmany = fscanf(stdin, "%s", op->in0);
      if (howmany != 1)
          Print error message and exit.
      break;
   case 1:   // param is in1
      howmany = fscanf(stdin, "%s" op->in1);
     if (howmany != 1)
          Print error message and exit.
      break;

    etc
   }

}
```

Using pointer (address) variables with non-structured types

A pointer is a *variable* whose value is the *address* of another variable.

The size of the pointer variable must be n bits where 2^n bytes is the size of the address space. For the Intel x86 and SPARC systems, address space size is 4GB and we have n=32.

The amount of space required to hold a pointer variable is always 4 bytes on these hardware platforms and is not related to the size of the entity to which it points.

We have previously described how to declare a pointer to a structure type:

```
image_t  *img;
```

and how to use it to access elements of the structure:

```
img->width = 600;
```

Pointers to basic types

A pointer to a basic type in defined in the same way.

```
int              *a;
short            *b;
unsigned char    *c;
int               x;
```

To assign the address of a variable of the appropriate type to the pointer to the pointer variable, specify the name of the pointer variable on the left and the address of variable pointed to on the right. The following assignment stores the address of *x* in *a.*

```
a = &x;      // assign the address of x to pointer a
```

Unlike structured types, basic types don't have elements!! So *a->* is meaningless.

To refer to the entity to which the pointer points *prepend* *

```
*a = 132;  // assign the value 132 to x (since a now points to x)
```

In summary

- when we use a standalone *a* in a program, we are referring to the pointer itself.
- when we use a **a* in a program, we are referring whatever *a* points to..

Initialization of pointers

Like all variables pointers must be initialized before they are used.

```
/* p17.c */

/* Example of a common error: failure to intialize */
/* a pointer before using it.. This program is     */
/* is FATALLY flawed....                           */

main()
{
   int *ptr;

   *ptr = 99;
```


```
    printf("val of *ptr = %d and ptr is %p \n", *ptr, ptr);
}
```

But unfortunately, on Linux this program appears to work!

```
class/215/examples ==> p17
val of *ptr = 99 and ptr is 0x40015360
```

The program *appears* to work because the address 0x40015360 just happens to be a legal address in the address space of this program. Unfortunately, it may be the address of some other variable whose value is now 99!!!

This situation is commonly referred to as a *loose pointer* and bugs such as these may be very hard to find.

Modifying the program causes failure

We can convert the bug from latent by active by changing the location of the variable *ptr.*

Here we move the location of the pointer variable down the stack by declaring an array of size 200.

```
class/215/examples ==> cat p18.c
/* p17.c */

/* Example of a common error: failure to intialize */
/* a pointer before using it                       */

main()
{
   int  a[200];  // force ptr to live down in uninit land
   int  *ptr;

   printf("val of ptr is %p \n", ptr);

   *ptr = 99;

   printf("val of *ptr = %d and ptr is %p \n", *ptr, ptr);
}

class/215/examples ==> p18
val of ptr is (nil)
class/215/examples ==>
```

Note that in this case the *second printf() is not reached* because the program segfaulted and died when it illegally attempted to assign the value 99 to memory location 0 (nil).

Minimizing latent loose pointer problems

Never declare a pointer without intializing it in the declaration.

```
int *ptr = NULL;
or
int *ptr = 0;
```

Never use NULL as synonym for integer or float 0.

Using *gdb* to find the point of failure:

The *gdb* debugger is a handy tool for identifying the location at which a program failed.

To use the debugger it is necessary to compile with the -g option.

```
class/215/examples ==> gcc -g -o p18 p18.c
```

To start the debugger use the *gdb* command and specify the program name

```
class/215/examples ==> gdb p18
```

At the *(gdb)* prompt you will usually want to tell the debugger to halt the program when it reaches the start of the main() function. The b command is short for breakpoint and tells the debugger where to stop the program. After a function is entered, source code line numbers can be used to specify breakpoints.

```
(gdb) b main
Breakpoint 1 at 0x804833b: file p18.c, line 11.
```

To start the program use the *run* command:

```
(gdb) run
Starting program: /local/westall/class/215/examples/p18
```

When the program reaches a breakpoint gdb will tell you and display the *next* line of code to be executed.

```
Breakpoint 1, main () at p18.c:11
11          printf("val of ptr is %p \n", ptr);
```

Use the *next* command to execute a single source statement. The *next* command will treat a function call as a single statement and not single step into the function being called. If you want to single step through the function use the step command to step into it. The output of the printf() is intermixed with gdb's output and is shown in blue below.

```
(gdb) next
val of ptr is (nil)
13          *ptr = 99;
```

Entering *next* again will cause the program to execute the flawed assignment. *gdb* will show you the line that caused the error (line # 13).

```
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x08048357 in main () at p18.c:13
13          *ptr = 99;
```

The *where* command can show you where the failure occured (along with a complete function activation trace.

```
(gdb) where
#0  0x08048357 in main () at p18.c:13
#1  0x40049917 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

The *where* command will show you exactly what line the program died on.

The *list* command lets you view the lines of code surrounding the point of failure.

```
(gdb) list
8           int  a[200];  //force ptr to live in uninitland
9           int  *ptr;
10
11          printf("val of ptr is %p \n", ptr);
12
13          *ptr = 99;
14
15          printf("val of *ptr = %d and ptr is %p \n",
                      *ptr, ptr);
16      }
```

To print the value of a variable use the *print* command.

```
(gdb) print ptr
$1 = (int *) 0x0
```

Attempting to print what ptr points to reaffirms what the problem is:

```
(gdb) print *ptr
Cannot access memory at address 0x0
```

Use the *q (quit)* command to terminate gdb

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
class/215/examples ==>
```

Correct use of the pointer

In the C language, variables that are declared within any basic block are allocated on the run-time stack at the time the basic block is activated.

```
/* p19.c */

main()                Addresses of y and ptr
{
   int  y;                    0xbffff894
   int* ptr;                  0xbffff890
   static int a;

   ptr = &y;   // assign the address of y to the pointer


   *ptr = 99;  // assign 99 to what the pointer points to (y)

   printf("y = %d ptr = %p addr of ptr = %p \n", y, ptr, &ptr);

   ptr = &a;

   printf("The address of a is %p \n", ptr);
}

class/215/examples ==> p17
y = 99 ptr = 0xbffff894 addr of ptr = 0xbffff890
The address of a is 0x804958c
```

Note that *a* is heap resident and has an address far removed from the stack resident *y* and *ptr*.

Using a pointer to process an array of numbers

This program demonstrates and how to process a dynamically allocated array of *ints* using a pointer.

```
/* p24.c */

/* This program illustrates the use of pointers in */
/* reading and processing an array of integers     */
/* It also shows how to access command line args   */

/* An upper bound on the number of ints to be read */
/* must be specified on the command line..         */
/*    p2 1000                                      */

#include <stdio.h>

int main(
int argc,       /* number of command line args */
char* argv[])   /* array of pointers to  args   */
{
  int *base;   // points to start of the array
  int *loc;    // current array location
   int  max;     // maximum number of values to read in
   int  count;   // actual number of values read in
   int  largest; // largest number in the array
   int  i;

   max = 100;    // want to hold 100 ints maximum;
```

Allocating storage for the array of ints.

- The *malloc()* (memory allocate) function is can be used to dynamically allocate an area of memory to be used as an array.

  The parameter passed to *malloc()* is the size of the area to be allocated in bytes.

- Therefore, if we want to allocate space for 100 *ints*, we must pass 400 to *malloc()* because each *int* occupies 4 bytes.

- The *sizeof()* facility should always be used instead of coding the constant 4.

```
count = 0;
base = (int *)malloc(sizeof(int) * max);
loc = base;
```

Size of a single element

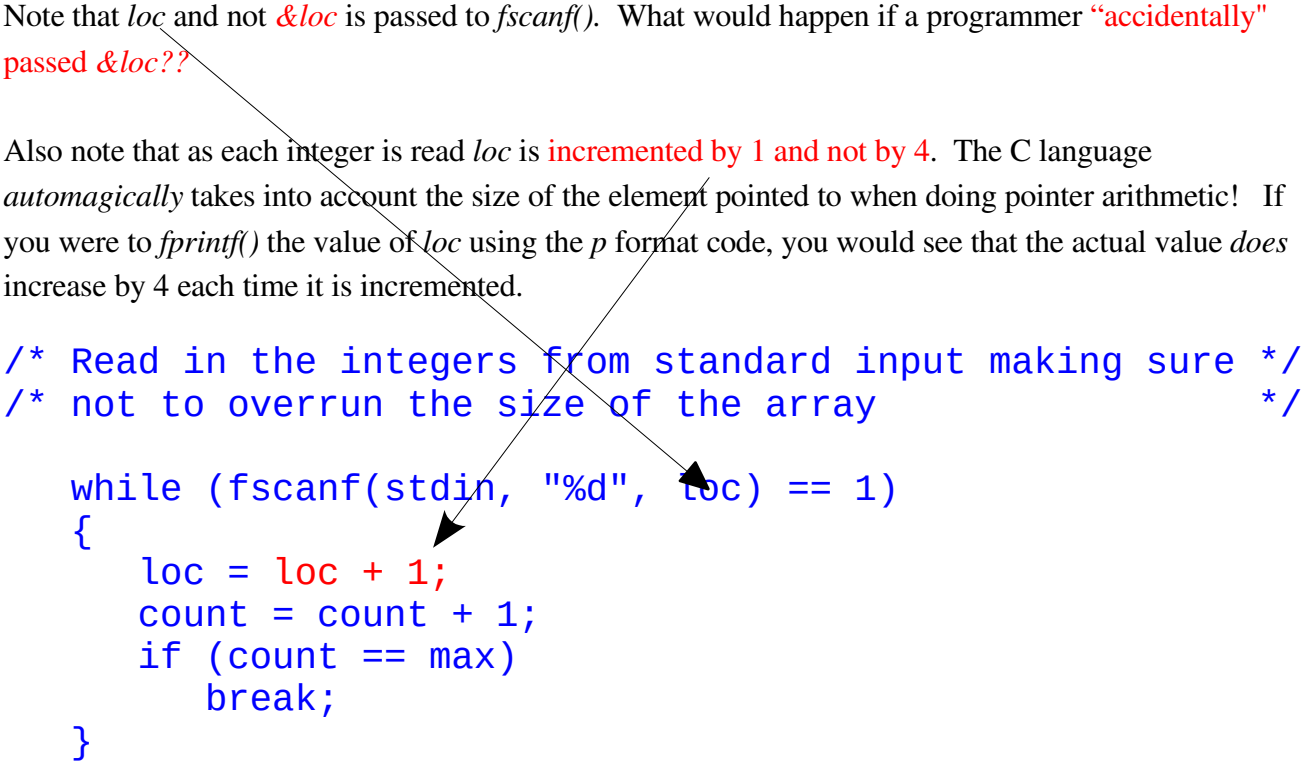
Number of elements (ints) to be allocated.

This operation is called a *cast.* It is used when a pointer to one type of entity is assigned to a pointer to another type of entity. Without it the C compiler will deliver a nasty warning message.

Reading the input values

Note that *loc* and not *&loc* is passed to *fscanf()*. What would happen if a programmer "accidentally" passed *&loc??*

Also note that as each integer is read *loc* is incremented by 1 and not by 4. The C language *automagically* takes into account the size of the element pointed to when doing pointer arithmetic! If you were to *fprintf()* the value of *loc* using the *p* format code, you would see that the actual value *does* increase by 4 each time it is incremented.



```
/* Read in the integers from standard input making sure */
/* not to overrun the size of the array                 */

   while (fscanf(stdin, "%d", loc) == 1)
   {
      loc = loc + 1;
      count = count + 1;
      if (count == max)
         break;
   }
```

You may also use array notation with a variable declared as a pointer:

```
   &base[count]    and
   (base + count)  and
   loc
```

are all three different names for the *same memory location!* Therefore, the read loop could also be written: (unless the directions tell you to use pointer notation)

```
   while (fscanf(stdin, "%d", &base[count]) == 1)
   {
      count = count + 1;
      if (count == max)
         break;
   }
```

Identifying the largest value in the array

Before starting the search for the largest number the value of *loc* is reset to point to the *base* of the array.

```
    loc = base;     // point loc to the start of the array
    largest = *loc; // init largest to the first value
    loc = loc + 1;

    i = 1;
    while (i < count)
    {
       if (largest < *loc)
          largest = *loc;

       loc = loc + 1;
       i = i + 1;
    }
    printf("Largest was %d \n", largest);
}
```

The *for* loop

The C language provides a useful mechanism for consolidating the *initialize, test, and increment* operations that are required when executing a loop a specified number of times.

```
    for (i = 1; i < count; i = i + 1)
    {
       if (largest < *loc)
          largest = *loc;

       loc = loc + 1;
    }
```

Functions that modify variables of their callers

We saw earlier that when a function tries to modify a parameter that is passed to it, it has no effect on the *callers* copy of that variable.

A caller *can* allow a function to modify its variables by passing a pointer to the value:

A function that swaps two integer values

```
int swap(
int *x,
int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

A caller of this function would operate as follows.

```
ndx = 0;
while (ndx < (count – 1))
{
   if (a[ndx] < a[ndx + 1])
     swap(&a[ndx], &a[ndx + 1]);
   ndx = ndx + 1;
}
```

Alternative notations

If *a* is an array or a pointer the names *&a[ndx]* and *a + ndx* are equivalent. Therefore the above code may be written using pointer notation as:

```
ndx = 0;
while (ndx < (count – 1))
{
   if (*(a + ndx) < *(a + ndx + 1))
     swap(a + ndx, a + ndx + 1);
   ndx = ndx + 1;
}
```

A fully pointerized version

The most efficient and readable way to construct a loop of this type is use a traveling pointer and avoid the array base address plus index computations altogether.

```
int *loc = a;    // same as a + 0 == &a[0]

for (ndx = 0; ndx < (count – 1); ndx = ndx+1)
{
   if (*(loc) < *(loc + 1))
     swap(loc, loc + 1);

   loc = loc + 1;

}
```

Two dimensional arrays in C

Two dimensional arrays are declared as follows:

```
double  x[4][5];
```

The declaration above creates 4 x 5 = 20 double precision values. Specific elements are accessed as follows:

```
x[2][3] = sqrt(10.0);
```

As with single dimension arrays the legal values of the first subscript are 0, 1, 2, 3 and the legal values of the second are 0, 1, 2, 3, 4

The values are stored in the following order:

```
x[0][0],  x[0][1], ... x[0][4],  x[1][0], .., x[3][4]
```

This is consistent with subscripting techniques commonly used in mathematics in which the first subscript commonly represents a *row* and the second represents a *column.*

Using array names as pointers

Recall that if we create a one dimensional array:

```
int a[5];
```

we can use the name *a* as a synonym for *&a[0]*.

In an analogous way the name *x[i]* is a synonym for *&x[i][0]*.

Thus the value of either *x[i]* *or* *&x[i][0]* *is* the address of the *ith* row of the array.

Reading and writing of 2 D array contents:

In this example we read in a 3 x 3 array reading one entire row with each read:

```
double tab[3][3];
int howmany = 0;
int i;

for (i = 0; i < 3; i++)
{
   howmany += fscanf(stdin, "%lf %lf %lf",
                  &tab[i][0], &tab[i][1], &tab[i][2]);
}
```

or equivalently

```
double tab[3][3];
int howmany = 0;
int i, j;

for (i = 0; i < 3; i++)
{
   for (j = 0; j < 3; j++)
   {
      howmany += fscanf(stdin, "%lf", &tab[i][j]);
   }
 }
```

The name *tab[i]* represents a pointer to the *ith* row of the array. So the following form will also work.

```
int howmany = 0;
double tab[3][3];
int i;

for (i = 0; i < 3; i++)
{
   howmany += fscanf(stdin, "%f %f %f",
                  tab[i], tab[i] + 1, tab[i] + 2);
}
```

Reading and writing of 2 D array contents:

```
double tab[3][3];
int i;

for (i = 0; i < 3; i++)
{
   fprintf(stderr, "%6.2lf %6.2lf %6.2lf \n",
                tab[i][0], tab[i][1], tab[i][2]);
}
```

To again print the array with 3 elements per line the doubly nested loop may be used but care must be taken to *print the newline in the correct spot.*

```
double tab[3][3];
int i, j;

for (i = 0; i < 3; i++)
{
   for (j = 0; j < 3; j++)
   {
      fprintf(stderr, "%6.2lf", tab[i][j]);
   }
   fprintf(stderr, "\n");
}
```

Arrays as operators

Arrays are often used as functions that transform vectors in two dimensional or three dimensional space. In fact *any* linear function on an n-dimensional vector space may be represented as multiplication by an *n x n* array where multiplication is defined as follows:

matrix operator		*vector operand*				*result*
0 1 2		1		0 x 1 + 1 x 2 + 2 x 3		8
3 4 5	x	2	=	3 x 1 + 3 x 2 + 5 x 3	=	24
6 7 8		3		6 x 1 + 7 x 2 + 8 x 3		44

This operation of multiplying a vector by a matrix may be coded as follows:

```
void xform(
double oper[3][3],  // operator matrix
double *opnd,       // operand vector
double *result)     // result vector
{
    int i, j;

    for (i = 0; i < 3; i++)      // i is row index
    {
        *(result + i) = 0;       // initialize sum!
        for (j = 0; j < 3; j++)  //j is col index
        {
           *(result + i) += oper[i][j] * *(opnd + j);
        }
    }
}
```

Color space transformations

Recall that in lab the NTSC standard for transmitting color TV signals was introduced. The component of interest in the lab was the *luminance.* The other two elements of the signal carry the color information and are called red and blue *chrominance.* For historical reasons the luminance is commonly referred to as Y and the chrominance values as Cb Cr or simply U V. The transformation from RGB space to YUV space is linear and thus can be represented by a 3 x 3 array.

```
float _RGBtoYUV[3][3] =
{
  { 0.299,   0.587,    0.114},    // standard luminance
  {-0.147,  -0.289,    0.436},    // blue - red - green
  { 0.615,  -0.515,   -0.100},    // red - blue - green
};
```

Decoding the above transformation we see that when it is applied to an (R, G, B) vector the resulting (Y, U, V) components are as shown below. Note that U = Cb postitively weights blue by negatively weights red and green, but V = Cr positively weights red but negatively weights G and B.

The transformation is called *linear* because Y, U, and V depend on R, G, and B to the first power – no squares or square roots for example. Also note that for U and V the *sum of the negative weights is exactly equal to the sum of the positive weights.* Thus the U value ranges from -0.436 * 255 to 0.436 * 255.

```
Y =  0.299 R + 0.587 G + 0.114 B
U = -0.147 R – 0.289 G + 0.436 B
V =  0.615 R  -0.515 G – 0.100 B
```

A useful way to do blue screen compositing is to convert (R, G, B) to (Y, U, V) and then check to see if (U - V) exceeds a certain threshold value. If so the pixel is assumed to come from the blue screen background .

The inverse transformation maps a YUV value back to RGB space. Needless to say, if you start with a specific RGB value, map it to YUV space and then map it back to RGB space, you should get the RGB value that you started with.

```
float _YUVtoRGB[3][3] =
{
   {1.000,  0.000,   1.140},
   {1.000, -0.395,  -0.581},
   {1.000,  2.032,   0.000},
};
```

The inverse transformation from YUV space to RGB space is shown below. Note that red and blue depend only upon Y and the V and U chrominance values respectively.

```
   R = 1.000 Y + 0.000 U  + 1.140 V
   G = 1.000 Y – 0.395 U  - 0.581 V
   B = 1.000 Y + 2.032 U  + 0.000 V
```

General Input and Output

The C *language* itself defines *no facility* for I/O operations. I/O support is provided through two collections of *mutually incomptible* function libraries

Low level I/O

```
open(), close(), read(), write(), lseek(), ioctl()
```

Standard library I/O

```
fopen(), fclose()              - opening and closing files
fprintf(), fscanf()            - field at a time with data conversion
fgetc(), fputc()               - character (byte) at a time
fgets(), fputs()               - line at a time
fread(), fwrite(), fseek()     - physical block at a time
```

Our focus will be on the use of the standard I/O library:

Function and constant definitions are obtained via `#include` facility. To use the standard library functions:

```
#include <stdio.h>
#include <errno.h>
```

Reading and writing disk resident files

So far, we have used *fscanf()* and *fprintf()* to access only the pre-declared files *stdin, stdout,* and, *stderr.* We can also define our own private file pointers (*FILE **)and bind them to disk files with the *fopen()* function. After that has been done, all file operations that can be performed on *stdin, stdout,* and *stderr* can also be performed directly on our disk file.

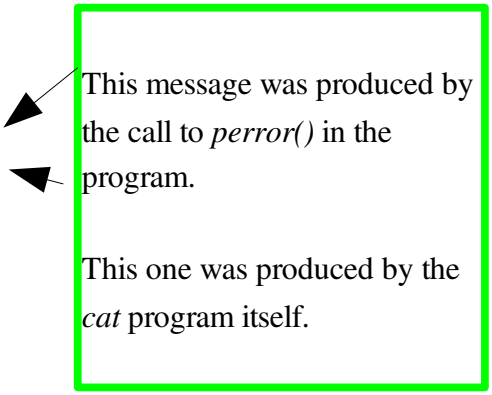
```
#include <stdio.h>
int main()
{
    FILE *f1;
    FILE *f2;
    int x;

    f1 = fopen("in.txt", "r");
    if (f1 == 0)
    {
        perror("f1 failure");
        exit(1);
    }
    f2 = fopen("out.txt", "w");
    if (f2 == 0)
    {
        perror("f2 failure");
        exit(2);
    }
    if (fscanf(f1, "%d", &x) != 1)
    {
        perror("scanf failure");
        exit(2);
    }
    fprintf(f2, "%d", x);
    fclose(f1);
    fclose(f2);
}
```

Examples of the use of the *perror()* function

In this example, we have not yet created *in.txt*

```
class/215/examples ==> gcc -o p6 p6.c
class/215/examples ==> p6
f1 failure:: No such file or directory
cat: in.txt: No such file or directory
```



Here, *in.txt* is created with the *cat* command:

```
class/215/examples ==> cat > in.txt
99
```

Now if we rerun *p6* and *cat out.txt* we obtain the correct answer

```
class/215/examples ==> p6
class/215/examples ==> cat out.txt
99
```

Note: The macros *scanf()* and *printf()* may be used as abbreviations for:

fscanf(stdin, ...) and
fprintf(stdout, ...)

Character at a time input and output

The *fgetc()* function can be used to read an I/O stream one byte at a time.
The *fputc()* function can be used to write an I/O stream one byte at a time.

Here is an example of how to build a *cat* command using the two functions. The *p10* program is being used here to copy its own source code from standard input to output.

```
class/215/examples ==> p10 < p10.c

/* p10.c */

#include <stdio.h>

main()
{
   int c;

   while ((c = fgetc(stdin)) > 0)
      fputc(c, stdout);
}
```

Even though *fgetc()* reads one byte at a time it returns an *int* whose low order byte is the byte that was read.

While *fputc()* and *fgetc()* are fine for building interactive applications, they are *very inefficient*. and should *never* be used for reading or writing a large volume of data such as a photographic image file.

Line at a time input with fgets().

Recall that *fscanf()* with the %s format code can be used to consume whitespace delimited strings. The *fgets()* function consumes *entire lines* at a time.

The *fgets(buffer_address, buf_size, file)* function can be used to read from a *stream* until either:

1 - a newline character *'\n' = 10 = 0x0a* is encountered or
2 - the specified number of characters - 1 has been read or
3 - end of file is reached.

There is no *string* data type in C, but a standard convention is that a *string* is an array of characters in which the end is of the string is marked by the presence of a byte which has the value binary 00000000 (sometimes called the *NULL* character).

The *fgets()* function will append the *NULL* character to whatever it reads in.

Since *fgets()* will read in multiple characters it is not possible to assign what it reads to a single variable of type *char*.

- Thus the address a *buffer* must be passed (as was the case with *fscanf()),* and
- the buffer must be of size at least *buf_size*

String at a time output with *fputs()*

The *fputs()* function writes a *NULL* terminated string to a *stream* (after stripping off the *NULL*). This is exactly what *fprintf()* with the %s format code does, but *fputs()* may do it in a slightly more efficient way.

The following example is yet another *cat* program, but this one works one line at a time.

```
class/215/examples ==> p11 < p11.c 2> countem

/* p11.c */

#include <stdio.h>
#include <string.h>

main()
{
   unsigned char buff[1024];
   int  line = 0;

    while (fgets(buff, 1024, stdin) != 0)
   {
      fputs(buff, stdout);
      fprintf(stderr, "%d %d \n", line, strlen(buff));
      line += 1;
  }
}
```

Here is the output that went to the standard error. Note that each line that appeared empty has length 1 (the newline character itself) and the lines that appear to have length 1 actually have length 2.

```
class/215/examples ==> cat countem
0 12
1 1
2 19
3 20
4 1
5 7
6 2
7 24
8 18
9 1
10 24
11 18
 :
20 2
```

Block input and output

The *fread()* and *fwrite()* functions are the most efficient way to read or write large amounts of data. The second parameter passed to the function is the *size of a basic data element* and the third parameter is the *number of elements*. Here the basic data element is a single byte so *1* is used. The *fread()* function returns the number of elements that it read.

Here is a still more efficient implementation of the *cat-like* program that copies standard input to the standard output.

```
class/215/examples ==> p12 < p12.c
/* p12.c */

#include <stdio.h>

main()
{
   unsigned char buff[1024];
   int  len = 0;
   int  iter = 0;


   while ((len = fread(buff, 1, 1024, stdin)) != 0)
   {
      fwrite(buff, 1, len, stdout);
      fprintf(stderr, "%d %d \n", iter, len);
      iter += 1;
  }
}
0 322
```

Exercises: Removing the parentheses surrounding

```
(len = fread(buff, 1, 1024, stdin))
```

will break the program. Explain exactly *how and why* things will go wrong in this case? What will happen if *len* in the *fwrite()* is replaced by 1024?

In summary -

Input

- If your application requires data conversion from ASCII text to internal binary or floating point you *must* use *fscanf()* with the %d or %f families of format codes.

- If you wish to consume one byte at a time including whitespace characters with no data conversion then *fgetc()* or *fscanf()* with the %c format code will work.

- If you wish to read consume input text a *word* at a time and store it as a NULL terminated string with with *whitespace* characters removed, then *fscanf()* with the %s format code is what you need.

- If you wish to consume input text one *line* at at time and store it as a NULL terminated string with with *whitespace* characters included, then *fgets()* will do the job.

- If you wish to read a fixed number of bytes with no data conversion, then use *fread().*

Output

- If your application requires data conversion from internal binary or floating point to ASCII text you *must* use *fprintf()* with the %d or %f families of format codes.

- If you wish to output one byte at a time with no data conversion then *fputc()* or *fprintf()* with the %c format code will work.

- If you wish to output a NULL terminated character array, then either *fprintf()* with the "%s" format code *or fputs()* can be used.

- If you wish to write a fixed number of bytes with no data conversion then *fwrite()* is the proper choice.

Scope and storage class of variables

The *scope* of a variable refers to those portions of a program wherein it may be accessed.

Failure to understand scoping rules can lead to two problems:

(1) Syntax errors (easy to find and fix)

(2) Accidentally using the wrong instance of a variable (sometimes very hard one to find).

Two general rules apply

(1) The *declaration* of a variable must *precede* any *use* of it.

(2) If a particular line of code is in the scope of *multiple variables of the same name the innermost declaration of the variable is the one that is used.*

Specific refinements of these rule include:

(1) the scope of any variables declared outside any function is all code in the source module that appears *after* the definition.

(2) the scope of any variable declared inside a basic block is all code *in that block and any blocks nested within that block that appears after* the definition.

Improper definition location

In this example the variable *y* is properly declared, but it is used before it is declared.

```
 1 /* p13.c */
 2
 3 /* this program demonstrates some of the characteristics */
 4 /* of variable scoping in C.                             */
 5
 6 /* The scope of y and z is all lines that follow their  */
 7 /* definitions.  Thus z may be used in f1 and f2        */
 8 /* but y may be used only in f2                         */
 9
10 int z = 12;
11
12 int f1(
13 int x)
14 {
15    x = x + y + z;
16    return(x);
17 }
18
19 int y = 11;
20
21 int f2(
22 int x)
23 {
24    x = x + y + z;
25
26 }
27
```

```
class/215/examples ==> gcc p13.c
p13.c: In function `f1':
p13.c:15: `y' undeclared (first use in this function)
p13.c:15: (Each undeclared identifier is reported only once
p13.c:15: for each function it appears in.)
p13.c: At top level:
p13.c:19: `y' used prior to declaration
```

Overlapping scope

Example program p14.c illustrates that multiple declarations of a variable having a single name is legal and results in overlapping scope.

In this program there do exist *three different variables named y.* When the program accesses *y* which *y* is used is governed by *the innermost definition rule*.

```
/* p14.c */

/* This program illustrates that multiple different */
/* declarations of a variable having the same name  */
/* may have overlapping scope.                      */

int y = 11;

int main( )
{
   int y = 12;

   if (1)
   {
      int y, z;
      y = 92;
     printf(“inner y = %d \n”, y);
   }
   printf("middle y = %d \n", y);
}
```

```
class/215/examples ==> p14
inner y = 92
middle y = 12
```

For sane debugging *never* use multiple variables with the *same name* and *overlapping scope*.

Note that if you always call your loop counter variable *i* and you always declare it at the *start of each function,* you are not violating this guidelines. In this case there are multiple *i's* but their scopes *don't overlap.*

Storage class

The *storage class* of a variable the area of memory in which a variable is stored.

The two available areas are commonly referred to as the *heap* and the *stack*.

Stack resident variables include:

- parameters passed to functions
- variables declared inside basic blocks that are *not* declared *static*

Heap resident variables include:

- variables declared outside all functions
- variables declared inside basic blocks that are declared *static.*
- memory areas dynamically allocated with *malloc()*

Storage for declared heap resident variables is

- assigned at the time a program is loaded and
- remains assigned for the life of a program.

Storage allocated on the heap using *malloc()* remains allocated until

- either *free()* is called or
- the program ends.

Stack resident variables are

- created at entry to the basic block that contains them and
- may be overwritten at exit from the block.

Non-Persistence of stack resident variables

The example below appears to contradict the claim that storage is assigned to a stack resident variable only for the time in which the block is active.

> At first entry to *f1()* the variable *x* is unintialized.
> The variable *x* is set to 55 before returning from *f1()*
> The variable is still 55 at entry on the second call.

```
void f1(void)
{
   int x;

   printf("At entry to f1 x = %d \n", x);

   x = 55;
}

main()
{
   f1();
   f1();
}
```

```
class/215/examples ==> p15
At entry to f1 x = -1073743532
At entry to f1 x = 55
```

x is uninitialized at first entry to *f1()*

x appears to retain its value at sencond entry to *f1()*

Example p16.c shows that the claim was indeed true and it was only *bad* luck that made it appear otherwise.

```
/* p16.c */

void f1(void)
{
   int x;

   printf("At entry to f1 x = %d \n", x);
   x = 55;
}

void f2(void)
{
   int z;

   printf("At entry to f2 z = %d \n", z);
   z = 102;
}

main()
{
   f1();
   f2();
   f1();
}
```

```
class/215/examples ==> p16
At entry to f1 x = -1073743644
At entry to f2 z = 55
At entry to f1 x = 102
```

It can be observed from the output above that in this particular case the variable *y* in *f1()* and *z* in *f2()* are in fact occupying the *same physical storage*.

Details of stack allocation

Two registers (special single-word high speed memories inside the CPU chip) manage access to the stack. The (E)SP stack pointer register always points to the last item pushed onto the stack. Each time a new item is pushed. SP is decremented by one word and the item is stored at the location whose address is in SP.

The (E)BP base pointer register provides a fixed point of reference used for addressing during the execution of a function. The BP register always points to the location where the caller's BP register was preserved.

The following actions are performed by the *caller* during a function call:

- Push the parameters onto the stack in *reverse* order.
- Issue the *CALL* instruction which pushes the address of the instruction following the *CALL* and then jumps to the first instruction of the called function.

The *called* function always begins with the following actions:

- Push the caller's BP register onto the stack
- Copy the SP register into the BP register
- Allocate space for local variable by decrementing the SP register

```
/* p27.c */

int adder(
int a,
int b)
{
   int d;
   int e;
   d = a + b;
   e = d - a;
   return(d);
}
```

At entry to the function *adder* the stack is organized as follows

```
Parm - 2 (b)
Parm - 1 (a)
Return address   <- SP (Stack pointer register)
```

The compiler produces a prologue to the body of the function which looks like. The *ebp* register is known as the *base pointer* or the *frame pointer*. All stack resident variables are addressed using base/displacement addressing with *ebp* serving as the base.

```
adder:
        pushl   %ebp                ;save caller's frame ptr
        movl    %esp, %ebp          ;set up my frame pointer
        subl    $8, %esp            ;allocate local vars
```

After the prolog completes the stack looks as follows

```
(ebp + 12) Parm - 2 (b)
(ebp +  8) Parm - 1 (a)
(ebp +  4) Return address
(ebp +  0) Saved ebp          <- EBP  (Frame pointer)
(ebp -  4) local var (d)
(ebp -  8) local var (e)      <- ESP (Stack pointer)
```

```
 d = a + b;

movl    12(%ebp), %eax    ;load b into eax
addl    8(%ebp), %eax     ;add a  to eax
movl    %eax, -4(%ebp)    ;store sum at d

e = d - a;

movl     8(%ebp), %edx    ;load a into edx
movl    -4(%ebp), %eax    ;load d into eax
subl    %edx, %eax        ;subtract a from d
movl    %eax, -8(%ebp)    ;save result in e

return(d);

movl    -4(%ebp), %eax    ;copy d to return reg
leave                     ;Copies EBP to ESP
                                then POPS EBP
```

After the *leave* executes

```
(ebp + 12) Parm - 2 (b)
(ebp +  8) Parm - 1 (a)
(ebp +  4) Return address   <- SP

ret                       ;POPS EIP
```

The *static* and *extern* modifiers

The *static* and *extern* modifiers are used as follows:

```
static int  num;
extern int  extnum;
extern char trashcan[256];
```

The action of the *static* modifier is dependent upon the location of the declaration.

- When used inside the body of a function, *static* forces the variable to

  1 - reside on the *heap* instead of the *stack* and thus

  2 - safely retain its value across function calls

  3 - But the *scope* of the variable remains only the remainder of the basic block in which it is declared.

```
int public_val;

int adder(int a)
{
    static int sum;
    sum += a;
    return(sum);
}
```

Use of *static* on variables declared outside function bodies

The action of the *static* modifier is dependent upon the location of the declaration.

- When used outside the body of a function as in declaration of *private_val*, *static*

  1- limits the scope of the variable to *this* source module.

  2 - and the variable still remains on the *heap.*

```
static int private_val;

int adder(int a)
{
    static int sum;
    sum += a;
    return(sum);
}
```

Use of the *extern* modifier.

The *extern* modifier can be used to access a *public* variable that is declared in another *source module.* If, in another module, I need to access the variable *public_val* that as declared previously I can declare:

```
extern int public_val;
main()
{
     public_val = 15;
}
```

However, the use of the *static* declaration will defeat the ability of *extern* modifier to access the variable.

```
extern int private_val;

main()
{
     private_val = 15;
}
```

The two source files *will* compile correctly but the linker *ld* will fail because p34.c no longer publishes the address of *private_val;*

```
class/215/examples ==> gcc p34.c p35.c
/tmp/ccNrTn6K.o(.text+0x12): In function `main':
: undefined reference to `private_val'
collect2: ld returned 1 exit status
class/215/examples ==>
```

- To avoid naming conflicts when multiple programmers are working on a large multi-module system it is a good idea to make all declarations *static* except those explicitly agreed upon as being shared through the *extern* mechanism.

- The use of "global variables" (variables declared in one module and accessed in another via the extern mechanism) should be *minimized.* Most programs don't need and shouldn't use them at all.

Static applied to functions

The static modifier can also be applied to functions to make them invisible outside the module in which they live.

```
static int addit(int a, int b)
{

}
```

To avoid naming conflicts when multiple programmers are working on a large multi-module system it is a good idea to make all *function* definitions static except those that are explicitly identified as providing services to external modules.

Boolean (logical) and Bit operations

We now treat somewhat more formally the logical operations expressed in C as &&, ||, and !. These are sometimes referred to as *Boolean* operations because of the work of English mathematician George Boole (http://en.wikipedia.org/wiki/George_Boole) who published foundational papers in the mid 1800's linking logic and algebra.

In Boolean logic the value 0 conventionally means *false* and 1 means *true*. *Boolean variables* are variables that take on values *false (0)* or *true(1)*. *Boolean functions* are functions of one or more input Boolean values whose output is also a boolean variable.

There are *four* distinct functions of a single Boolean variable. The functions are described using a *truth table*. *f0* is the *false* function, f1 the *identity* function, *f2* the *not* function and *f3* the *true* function. Note that the function identifier 0, 1, 2, 3 is simply the decimal representation of the binary values taken on by the function when read top to bottom. (The values of *f2(x)* are 1 0 *and* 10 binary is 2 decimal).

x	f0(x)	f1(x)	f2(x)	f3(x)
0	0	0	1	1
1	0	1	0	1

In the C language, the *not* function is implemented via the *!* operator. The statement

```
printf("%d \n",  !(2 < 1));
```

will cause a value of 1 to be printed. The relation 2 < 1 is false and so !(2 < 1) is true.

Boolean functions of two variables

There are 16 different functions of 2 boolean variables. Three of them that are commonly used in making logical inferences are boolean functions *or, and*, and *xor*.

and: the result is true if and only if both operands are true
or: the result is true if either operand is true
xor: the result is true if and only if exactly one of the operands is true.

		f7(x, y)	*f1(x, y)*	*f6(x, y)*
x	***y***	***x or y***	***x and y***	***x xor y***
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

All 16 of the functions may be enumerated by creating a column containing the binary representations of the values {0, 1, 2, .. 15}

Example
Complex logical inferences are easier to evaluate algebraically than "logically".

Example: Is the following sentence true or false!

Dr. Westall is old, or (it is not the case that (it is raining and the minimum project score is 100))) or (Dr. Westall is not old, or it is not raining).

Let x = Dr. Westall is old (T) x = 1
y = It is raining right now (F) y = 0
z = The minimum score on the project is 100 (F) z = 0

(x or not(y and z)) or (not x or not y)
(T or not (F and F)) or (not T or not F)
(T or not F) or (F or T)
(T or T) or (F or T) = T or T = T

Proofs via truth tables:

Prove: not (x and y) == not x or not y

x	y	!x or !y	x and y	!(x & y)
0	0	1	0	1
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

Boolean operations in C

In C there are no Boolean variables.

0 corresponds to False (0)
not zero corresponds to True (1)

There are three Boolean functions

not - *!*
and - *&&*
or - ||

- Input values to these functions may be integer or floating point. A value of 0 is FALSE and any other value is TRUE.

- The output value is an integer 0 or 1.

Bit operations

Four of the bit operators are Boolean functions performed in *bitwise fashion on all of the bits* of an integer data type ((unsigned) char, short, int, or long).

These operators are:

~	*not*
|	*or*
&	*and*
^	*xo*r

```
~10101010 = 01010101

   10101010
 & 11110000
   10100000

   10101010
 | 11110000
   11111010


   10101010
 ^ 11110000
   01011010
```

Bitwise and boolean operators are *not interchangable*

5 & 2 = 0
5 && 2 = 1

The shift operators

>> *shift bits toward least significant bit*
<< *shift bits toward most significan bit*

Bits shifted out of the value are lost
Bits shifted into the *lsb* position during << always are 0.
Bits shifted into the *msb* position during >> depend on signed/unsigned int.

signed => current msb is duplicated
unsgined => a 0 is injected.

01110001 >> 2 = 00011100
01110001 << 2 = 11000100

Example - reading a binary number

This function is the binary analog of fscanf() with the %d, %o, or %x format codes. It reads ASCII character representations of numbers expressed in binary and converts them to binary integers. For example, If the input is:

1011

the value produced should be 11 = 1 x 2^3 + 1 x 2^1 + 1 x 2 ^ 0

It is necessary to consume the characters one at a time. Each time a 1 or 0 is consumed the result is shifted left one bit position and the value just consumed is or-ed into the low order bit position.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int bscanf(
FILE *input,
int  *value)
{
   int c;
   int result = 0;
   int found = 0;

/* Consume any leading whitespace.  The isspace()  */
/* function returns true if c is a "whitespace"    */
/* character.                                      */

   do
      c = fgetc(input);
   while (isspace(c));

/* On exit from this loop c should hold the first digit */
```

```
/* Now read one character at a time – we contine while  */
/* when c > 0 ( not end of file) and a 0 or 1 value     */
/* is read                                              */

   while ((c > 0 )  && (c == '0' || (c == '1')))
   {
      result <<= 1;          // make room for new bit
      result |=  c – '0';    // insert new bit
      c = fgetc(input);      // get next character
      found = 1;
   }
   *value = result;
   return(found);
}
```

Example - printing a binary number

This function is the binary analog of fprintf() with the %d, %o, or %x format codes

```
int bprintf(
FILE *output,
int  value)
{

/* We can test a specific bit in an int by anding it */
/* with a "mask" that has a 1 in the specific bit    */
/* position.                                         */

   unsigned int mask = 0x80000000;

/* We want to skip over non-significant leading zeros */
/* Don't print 00000000000000000000000000001011       */
/* Do    print 1011
                                  */
/* So we start looking at the msb and continue until  */
/* a 1 bit is found.                                  */

   while (!(mask & value))
      mask >>= 1;

/* Found a 1 bit.. print remaining bits as '0' or '1' */

   while (mask)
   {
      (mask & value) ? fputc('1', output):
                       fputc('0', output);
      mask >>= 1;
   }
   return(1);
```

Malloc'd objects, garbage collection, and memory leaks

Some languages (e.g. Java) provides for "automagic" garbage collection in which storage for an object is magically reclaimed when all references to an object have gone out of existence.

C provides no such mechanism.

A *memory leak* is said to have occurred when:

1. the last pointer to a malloc'd object is reset or
2. the last pointer to a malloc'd object is a local variable in a function from which a return is made,

In these cases the *malloc'd* memory is no longer accessible. Excessive leaking can lead to poor performance and, in the extreme, *program failure*.

Therfore C programmers must recognize when last pointer to *malloc'd* storage is about to be lost and use the *free()* function call to release the storage before it becomes impossible to do so.

Examples of incorrect pointer use and memory leaking are commonly observed in student programs

Problem 1: The instant leak.

This is an example of an *instant leak*. The memory is allocated at the time *temp* is declared and leaked when *temp* is reassigned the address of the first object in the list.

```
obj_t *temp = malloc(sizeof(obj_t));

temp = list->head;
while (temp != NULL)
  -- process list of objects --
```

Two possible solutions:

Insert *free(temp)* before *temp = list->head;*

This eliminates the leak, but what benefit is there to allocating storage and instantly freeing it???

Change the declaration to *obj_t *temp = NULL;*

This is the correct solution.

A rational rule of thumb is *never malloc memory unless you are going to write into it!*

Another good rule of thumb is to *never declare a pointer without initializing it.*

Problem 2: The traditional leak

Here storage is also allocated for *univec* at the time it is declared.

The call to *vl_unitvec3()* writes into that storage.

If the storage is not *malloc'd,* then *univec* will not point to anything useful and the call to *vl_unitvec3()* will produce a segfault or will overwrite some other part of the program's data. So this *malloc()* is necessary.

```
{
    double *univec = malloc(3 * sizeof(double));

    vl_unitvec3(dir, univec);
           :
    more stuff involving univec
           :
    return;
 }
```

However, the instant the return statement is executed, the value of *univec* becomes no longer accessible and the memory has been leaked.

Here the correct solution is to add

```
  free(univec);
```

just before the return;

A rational rule of thumb is: *malloc'd storage must be freed before the last pointer to it is lost.*

Problem 3: Overcompensation

The concern about leakage might lead to an overcompensation. For example, an object loader might do the following:

```
{
     obj_t *new_obj;
           :
     new_obj = malloc(sizeof(obj_t));
          :
     if (list->head == NULL)
     {
          list->head = list->tail = new_obj;
     }
     else
     {
          list->tail->next = new_obj;
          list->tail = new_obj;
     }
     free(new_obj);

     return(0);
}
```

This problem is the reverse of a memory leak. A live pointer to the object exists through the *list* structure, but the storage has been *freed.*

The results of this are:

1. Usually attempts to reference the freed storage will succeed.
2. The storage will eventually be assigned to another object in a later call to *malloc().*
3. Then "both" objects will occupy the same storage.

Rational rule of thumb: *Never free an object while live pointers to the object exist.* Any pointers to the freed storage that exist after the return from *free()* should be set to NULL.

To fix this problem the *free(new_obj) must be deleted from the code*. If the objects in the object list are to be freed, *it is safe to do so only at the end of the raytrace.*

It is not imperative to do so at that point because the Operating System will *reclaim all memory* used by the process when the program exits.

Problem 3b: Overcompensation revisited

The *free()* function must be used *only* to free memory previously allocated by *malloc()*

```
unsigned char buf[256];
:
:
free(buf);
```

is a fatal error.

The *free()* function must be not be used to free the same area twice.

```
buf = (unsigned char *)malloc(256);

  :
  free(buf);

:
free(buf);
```

is also fatal.

The general solution: Reference counting

For programs even as complicated as the raytracer it is usually easy for an experienced programmer to know when to *free()* dynamically allocated storage.

In programs as complicated as the Linux kernel it is not.

A technique known as *reference counting* is used.

```
typedef struct obj_type
{
   int refcount;
    :
  :
} obj_t;
```

At object creation time:

```
     new_obj = malloc(sizeof(obj_t));
     new_obj->refcount = 1;
```

When a new reference to the object is created

```
     my_new_ptr = new_obj;
     my_new_ptr->refcount += 1;
```

When a reference is about to be reused or lost

```
     my_new_ptr->refcount -= 1;
     if (my_new_ptr->refcount == 0)
        free(my_new_ptr);
     my_new_ptr = NULL;
```

In a multithreaded environment such as in an OS kernel it is *mandatory* that the testing and update of the reference counter be done *atomically*. This issue will be addressed in CPSC 322.