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 less fast than main memory
- non-volatile (unless you drop the computer on the floor!)

Communication elements – transfer data items between processing and/or 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 modeled by 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

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 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. In this case 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 110

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 here 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)* which tells the human that

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

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 100 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 100,200       !Store the upper limit in box 200
1      store 0,100         !Initialize the sum to 0
2      store 1,101         !Initialize the value to be added
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,200,4  !as long as the value in box 101 is less than or equal to the 100 in box
                           !200 continue to add
7      halt
```

Exercise for next time: 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. (You may assume both values are integers)

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* 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)*

The human computer and the C programming language:

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

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

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 (counting) numbers (signed (positive and negative) and unsigned(non-negative))
- Floating point numbers (those with a fractional part often shown in scientific notation)
- The “Latin” character set in which English is written

Before turning to the details of the representation, we should 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 values which are commonly called {0 , 1}.

Because a single bit encodes so little information, 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 table of individually addressable storage elements (analogous to the 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 *"box" must contain more than 1 bit* of information.

The basic addressable unit of memory

The "optimal" number of bits in the "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.

A single 8-bit byte can encode 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 storage element
the *contents* of a storage element

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.

Aggregation of basic memory elements

8 bit bytes (or “mailboxes”) are useful for encoding

- the identity of domestic animals commonly found in the home
- characters of the Latin alphabet in which English is written
- ... and many other things, but ....

but since a byte can represent only 256 different values it is not very useful for numerical applications requiring high precision.

Representing numeric entities

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 7 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 arithmetic, they do so by discarding any remainder
- So 100 / 9 = 10 (instead of 10.111111..........), and 5 / 2 = 2 (instead of 2.5).

Positional number systems

We have seen that integers are stored in a computer *words* 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 the number of a particular digit within a number determines the digit's contribution to the final value of the number.

Roman numerals 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*.

Exercise: Convert the Roman number XLVII to decimal.

Binary or base 2 arithmetic is a positional system that works just like 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 ^ n*.

Binary representation

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

Thus, the binary number 10110 means

```
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.

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:

```
   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

Exercise: 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 which 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
8	3
16	4

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 base 16 numbers. Thus we augment the digits with the first 6 letters of the alphabet.

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

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

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

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

```
  Base 2      Base 16
1001 0011  =    93
```

Unsigned and signed integer arithmetic

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

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.

Unsigned 8 bit arithmetic:

Possible values in base 10 are 0, 1, 2, ... 255

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!!

Signed 8 bit arithmetic -

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 + twos_complement(b)*
- Unlike “sign bit” systems, the twos complement system has only a single 0

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!
```

Encoding the alphabet

Its useful to encode text in computer systems 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 of a code:

- 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.

```
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 digits is in the range hex 30 to hex 39

```
48 30 0
49 31 1
 :
56 38 8
57 39 9
```

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

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

The upper case letters are next

```
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

```
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

```
 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*.

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*. Exactly *one* of the function must have the name *main()*.

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

```
 int main(void)
```

```
 {
    int ret_val;

    ret_val = 12;
    return(ret_val);
 }
```

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* 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 integer variables in the C language

A C variable is declared via a statement of the following form:

optional_modifier type_name variable_name;

The following type names create integer values. The default modifier is *signed* . To create an *unsigned* integer the *unsigned* modifier must be explicitly provided.

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

Example declarations:

unsigned char	*value1;*
int	*xyz;*
unsigned short	*has16bits;*

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, void, while, char, unsigned ....) See appendix A of text*
- *Should not be* the name of a standard function or variable.... but how do I know what those are?

Any signed or unsigned integer variable can hold

- values used in arithmetic computation
- ASCII encoded representation of characters

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

Executable code

Executable code is constructed from *expressions*

Expressions consist of (legal) combinations of:

- *operands which may be further classified as*
    - *constants*
    - *variables*
    - *function calls*

and

- *operators*

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

Integer variables may be declared as previously described (unsigned) char, short, int, long

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:

fscanf(stdin, “%d”, &input_num);

***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 on the right side to the *variable* on the left side.

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

Add, subtract, multiply, divide

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

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

Logical: `!, &&, ||`

Bitwise: `&, |, ~, ^`

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
y == 5
z - 10
```

Some operators also support either one or two operands

```
-x
```

Others support only one operand

```
!x
```

Expressions are built by combining operators and operands

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

What is the value of the above expression?

Suppose I tell you that the variable *v1* currently has value *4* and *v2* has value 3.

Now what is the value?

The C compiler has a set of immutable rules for evaluating such expressions. The rules are called *precedence and associativity rules* but they are sometimes 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 *innermost parentheses first*.

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

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

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

Exercise: 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;
```

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 with no arguments as follows:

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

C programs often rely on functions that reside in the *C run time library.* 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 library, *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:

Comments should be used to describe the use of variables

```
#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);
}
```

If *fscanf()* succeeds, 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 which is binary 1 0000 0101 or hex 105 = 1 * 16 ^ 2 + 5. The *fscanf()* function performs this conversion *automagically* when the %d format code is used.

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.

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()* where the addresss 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!

Managing flow of control in high level languages

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

```
        if (some-condition-is-true)
            goto statement_label;

        some-statement1;
        some-statement2;

statement_label:
        x = y + 2;
```

If "some condition" is true, then *some-statement1* and *some-statement2* will not be executed.

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 should have the ability change the value of expression!!!*

Exercise: What are the ways in which a while statement whose target does not modify the basic block may fail?

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.

The complete program

The following program will process an arbitrary number of input pairs and terminate when the end of the input file is reached:

```
acad/cs101/notes06 ==> cat p4.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);

   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 format codes

Additional *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;
   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 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
sum variables
previous input values

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;    // the indicator variable; 1 => found a 13
   int value;      // the value just read
   int howmany;    // 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.

```
Sample input:
1 2 4 77 12 13
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 and Counting: Compute 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. It is initialized to have the value 0 and incremented each time a 3 is encountered in the file.

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

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

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

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

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

```
Sample input: 1 2 3 2 1 3 4
Sample output: 2

Sample input: 3 3 3
Sample output: 3
```

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;    // how many integers were read
   int minval;     // the minimum value seen so far
   int value;      // 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
```

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;         // the sum of the single digit numbers
   int value;       // the value just read
   int howmany;     // howmany values were read

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

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

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

```
Sample input: -11 12 -1 15
Sample output: 15

Sample input: 14 12 9
Sample output: 35
```

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

Here the program state that must be maintained is the *sum* of all values between (-10 and 10) 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.* Otherwise the value is ignored.

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

int main()
{
   int sum;         // the sum of the single digit numbers
   int value;       // the value just read
   int howmany;     // 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 was %d \n", sum);
   return(0);
}
```

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

Sample input: 14 12 99
Sample output: 0
```

Searching for sequences: 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.* 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;          // value before that
   int old;            // previous value
   int value;          // current value
   int howmany;        // 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()??* Unrecognized implicit assumptions regarding sane input are a leading cause of program failure!

The main loop illustrates several 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*
- the *&&* operator is used to combine conditions. The result is true if and only if *all* the subconditions are true.

```
   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
```

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;       // the current value in [1, 10000]
   int valsqr;    // square of the value
   val = 1;
   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 remember the strategy from the {13, 14, 15} problem in which we learned how to "remember" a window of 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

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.

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.

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. 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

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.

Controlling 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 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

Like the *stdout* the *stdin* may also be redirected.   To 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 *input.txt* and any data written to the *stdout* will end up in the file *output.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
```

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
```

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

```
    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 spaces) 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

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

Processing numeric values supplied on the command line

Suppose your 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
```

(Only the width is specified because flags have individual aspect ratios)

```
#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);
}
```

Other control flow 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];
}
```

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* and *gimp.*
- Other well known formats include JPEG (.jpg), TIFF (.tif), GIF (.gif), and PNG (.png)

We will 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
- (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

***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 arguments                                         */
/*     width  in pixels                                    */
/*     height in pixels                                    */
/*     red value                                           */
/*     green value                                         */
/*     blue value                                          */

#include <stdio.h>

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

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

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

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

   fprintf(stdout, "P6\n");
   fprintf(stdout, "%d %d 255\n", width, height);

   while (count < (width * height))
   {
      fprintf(stdout, "%c%c%c", red, green, blue);
      count = count + 1;
   }
   return(0);
}
```

%c format prevents data conversion

%c format generates 1 byte of output

no blank spaces permitted in format string

Sample input file

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

Running the program with input and output redirection

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

Viewing the data with a hexadecimal / ASCII dump utility

```
class/101/examples ==> hexdump < pic1.ppm | more

        0 - 50 36 0a 32 30 30 20 31 35 30 20 32 35 35 0a c8
            P  6     2  0  0     1  5  0     2  5  5
       10 - 40 e0 c8 40 e0 c8 40 e0 c8 40 e0 c8 40 e0 c8 40
            @        @        @        @        @        @
       20 - e0 c8 40 e0 c8 40 e0 c8 40 e0 c8 40 e0 c8 40 e0
                  @        @        @        @        @
       30 - c8 40 e0 c8 40 e0 c8 40 e0 c8 40 e0 c8 40 e0 c8
               @        @        @        @        @
```

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
```

$$\begin{array}{rr} & 15 \text{ (header length)} \\ 3\text{x } 200 \text{ x } 150 = & \underline{90000 \text{ (image data) }} \\ & 90015 \end{array}$$

This is the image: