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);
}
```

Input of integer values

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

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 1 0000 0101 or hex 105 = 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 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 to 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 pair!

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*.

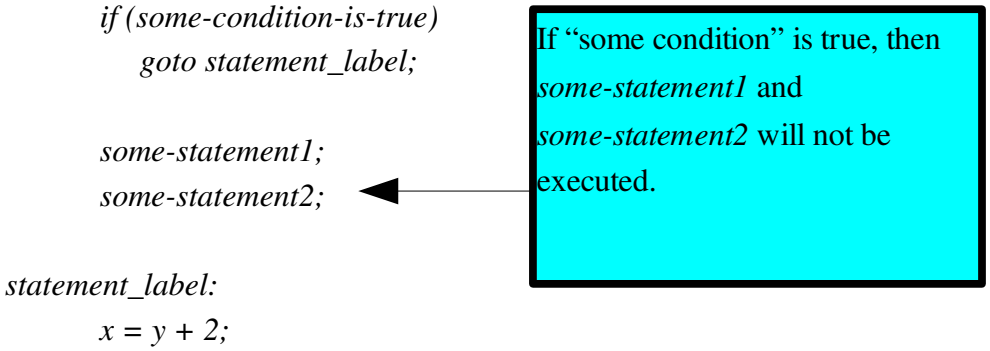
Recall the *human computer:*

- 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:



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..

Control flow in C programs

We will now introduce control flow in C programs, but before doing so, we will present a *slightly* formal view of the structure of a C program so that we don't have to learn *exclusively* by example.

A C program consists of a collections of one or more functions. Exactly one of the function must have the name *main()*.

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

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

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

```
{
        declaration of variables
        executable code
}
```

For example:

```
{
   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;

}
```

Executable code

Executable code is constructed from *expressions*

Expressions consist of (legal combinations of):

constants
variables
operators
function calls

Integer *constants* may be expressed as

decimal numbers:	71	must *not* start with 0!!!
octal numbers:	0107	always start with 0
hexadecimal numbers	0x47	always start with 0x
ASCII characters	'G'	

All of the above constants have the same value!

Different encodings may be freely intermixed in expressions

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

Operators consist of several classes. For now, we will be using only the ones shown in *red.*

Arithmetic:	`+, -, *, /, %`
Comparative:	`==, != , <, <=, >, >=`
Logical:	`!, &&, ||`
Bitwise:	`&, |, ~, ^`
Shift:	`<<, >>`

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

fscanf(stdin, "%d", &input_num)

An example expression

```
v1 + 5 * v2 / 3 * sqrt(16.0)
```

What is the value of the above expression?

Suppose I tell you that the variable *v1* currently has value *7* and *v2* has value 2.

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* 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 *inside out*.

```
v1 + ((5 * v2) / (3 * sqrt(16.0)))
```

```
v1 + (5 * (v2 / 3)) * sqrt(16.0)
```

```
((v1 + 5) * v2) / (3 * sqrt(16.0))
```

Exercise: Find the value of each of the above three expressions. For now, assume (incorrectly) that the sqrt() function returns an integer.

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;

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 *basic block.*

It should be clear that the *statement* or the *basic-block should have the ability change the value of expression!!!*

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

```
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.